


# 1 NumLin: Linear Types for Linear Algebra

2 Dhruv C. Makwana 

3 Unaffiliated [dhruvmakwana.com](http://dhruvmakwana.com)

4 [dcm41@cam.ac.uk](mailto:dcm41@cam.ac.uk)

5 Neelakantan R. Krishnaswami 

6 Department of Computer Science and Technology, University of Cambridge, United Kingdom

7 [nk480@cl.cam.ac.uk](mailto:nk480@cl.cam.ac.uk)

## 8 — Abstract —

9 We present NUMLIN, a functional programming language whose type system is designed to enforce  
10 the safe usage of the APIs of low-level linear algebra libraries (such as BLAS/LAPACK). We do  
11 so through a brief description of its key features and several illustrative examples. We show that  
12 NUMLIN's type system is sound and that its implementation improves upon naïve implementations  
13 of linear algebra programs, almost towards C-levels of performance. By doing so, we demonstrate (a)  
14 that linear types are well-suited to expressing the APIs of low-level linear algebra libraries accurately  
15 and concisely and (b) that, despite the complexity of prior work on it, fractional permissions  
16 can actually be implemented using simple, well-known techniques and be used practically in real  
17 programs.

18 **2012 ACM Subject Classification** Theory of computation → Program specifications

19 **Keywords and phrases** numerical, linear, algebra, types, permissions, OCaml

20 **Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

21 **Supplement Material** [www.github.com/dc-mak/NumLin](http://www.github.com/dc-mak/NumLin)

## 22 1 Introduction

23 Programmers writing numerical software often find themselves caught on the horns of a  
24 dilemma. The foundational, low-level linear algebra libraries such as BLAS and LAPACK  
25 offer programmers very precise control over the memory lifetime and usage of vector and  
26 matrix values. However, this power comes paired with the responsibility to manually manage  
27 the memory associated with each array object, and in addition to bringing in the familiar  
28 difficulties of reasoning about lifetimes, aliasing and sharing that plague low-level systems  
29 programming; this also moves the APIs away from the linear-algebraic, mathematical style  
30 of thinking that numerical programmers want to use.

31 As a result, programmers often turn to higher-level languages such as Matlab, R and  
32 NumPy, which offer very high-level array abstractions that can be viewed as ordinary  
33 mathematical values. This makes programming safer, as well as making prototyping and  
34 verification much easier, since it lets programmers write programs which bear a closer  
35 resemblance to the formulas that the mathematicians and statisticians designing these  
36 algorithms prefer to work with, and ensures that program bugs will reflect incorrectly-  
37 computed values rather than heap corruption.

38 The intention is that these languages can use libraries BLAS and LAPACK, without  
39 having to expose programmers to explicit memory management. However, this benefit comes  
40 at a price: because user programs do not worry about aliasing, the language implementations  
41 cannot in general exploit the underlying features of the low-level libraries that let them  
42 explicitly manage and reuse memory. As a result, programs written in high-level statistical  
43 languages can be much less memory-efficient than programs that make full use of the powers  
44 the low-level APIs offer.



© Dhruv C. Makwana and Neelakantan R. Krishnaswami;  
licensed under Creative Commons License CC-BY

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:44

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

So in practice, programmers face a tradeoff: they can eschew safety and exploit the full power of the underlying linear algebra libraries, or they can obtain safety at the price of unneeded copies and worse memory efficiency. In this work, we show that this tradeoff is not a fundamental one.

NUMLIN is a functional programming language whose type system is designed to enforce the safe usage of the APIs of low-level linear algebra libraries (such as BLAS/LAPACK). It does so by combining linear types, fractional permissions, runtime errors and recursion into a small, easily understandable, yet expressive set of core constructs.

NUMLIN allows a novice to understand and work with complicated linear algebra library APIs, as well as point out subtle aliasing bugs and reduce memory usage in existing programs. In fact, we were able to use NUMLIN to find linearity and aliasing bugs in a linear algebra algorithm that was *generated* by another program *specifically designed to translate matrix expressions into an efficient sequence of calls to linear algebra routines*. We were also able to reduce the number of temporaries used by the same algorithm, using NUMLIN’s type system to guide us.

NUMLIN’s implementation supports several syntactic conveniences as well as a *usable* integration with real OCaml libraries.

## 1.1 Contributions

In this paper

- we describe NUMLIN, a linearly typed language for linear algebra programs
- we illustrate that NUMLIN’s design and features are well-suited to its intended domain with progressively sophisticated examples
- we prove NUMLIN’s soundness, using a step-indexed logical relation
- we describe a very simple, unification based type-inference algorithm for polymorphic fractional permissions (similar to ones used for parametric polymorphism), demonstrating an alternative approach to dataflow analysis [8]
- we describe an implementation that is both compatible with and usable from existing code
- we show an example of how using NUMLIN helped highlight linearity and aliasing bugs, and reduce the memory usage of a *generated* linear algebra program
- we show that using NUMLIN, we can achieve parity with C for linear algebra routines, whilst having much better static guarantees about the linearity and aliasing behaviour of our programs.

## 2 NumLin Overview and Examples

### 2.1 Syntax

NUMLIN’s concrete syntax is inspired by that of OCaml. It desugars (Figure 2) into a smaller, core type and expression grammar (Figure 1).

### 2.2 Type System and Other Features

The core type theory of NUMLIN is a nearly off-the-shelf linear type theory [3], supporting familiar features such as linear function spaces  $A \multimap B$  and tensor products  $A \otimes B$ . We adopt linearity – the restriction that each program variable be used at most once – since it allows us to express purely functional APIs for numerical library routines that mutate arrays

$$\begin{aligned}
f &::= 'fc \mid \mathbf{z} \mid f \ \mathbf{s} \\
t &::= \mathbf{unit} \mid \mathbf{bool} \mid \mathbf{int} \mid \mathbf{elt} \mid f \ \mathbf{arr} \mid f \ \mathbf{mat} \mid !t \mid 'fc. t \mid t \otimes t' \mid t \multimap t' \\
e &::= p \text{ (primitives)} \mid x \text{ (variable)} \mid \mathbf{let} \ x = e \ \mathbf{in} \ e' \mid () \mid \mathbf{let} \ () = e \ \mathbf{in} \ e' \mid \mathbf{true} \mid \mathbf{false} \\
&\quad \mathbf{if} \ e \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2 \mid k \text{ (integer)} \mid l \text{ (heap location)} \mid el \text{ (array element)} \\
&\quad \mathbf{Many} \ v \mid \mathbf{let} \ \mathbf{Many} \ x = e \ \mathbf{in} \ e' \mid \mathbf{fun} \ 'fc \rightarrow e \text{ (frac. perm. abstraction)} \\
&\quad e[f] \text{ (frac. perm. specialisation)} \mid (e, e') \mid \mathbf{let} \ (a, b) = e \ \mathbf{in} \ e' \\
&\quad \mathbf{fun} \ x : t \rightarrow e \mid e \ e' \mid \mathbf{fix} \ (g, x : t, e : t')
\end{aligned}$$

■ **Figure 1** Core fraction  $f$ , type  $t$  and expression  $e$  grammar of NUMLIN. Values  $v$  are a subset of the expressions, their full definition and a list of all primitives  $p$  is in Appendix E.

$$\begin{aligned}
x[e] &\Rightarrow \mathbf{get} \ \_ \ x \ (e) && \text{(similarly for matrices)} \\
x[e_1] := e_2 &\Rightarrow \mathbf{set} \ x \ (e_1) \ (e_2) && \text{(similarly for matrices)} \\
pat &::= () \mid x \mid !x \mid \mathbf{Many} \ pat \mid (pat, pat) \\
\mathbf{let} \ !x = e_1 \ \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ \mathbf{Many} \ x = e_1 \ \mathbf{in} \\
&\quad \mathbf{let} \ \mathbf{Many} \ x = \mathbf{Many} \ (\mathbf{Many} \ x) \ \mathbf{in} \ e_2 \\
\mathbf{let} \ \mathbf{Many} \ \langle pat_x \rangle = e_1 \ \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ \mathbf{Many} \ x = x \ \mathbf{in} \\
&\quad \mathbf{let} \ \langle pat_x \rangle = x \ \mathbf{in} \ e_2 \\
\mathbf{let} \ (\langle pat_a \rangle, \langle pat_b \rangle) = e_1 \ \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ (a, b) = a\_b \ \mathbf{in} \ \mathbf{let} \ \langle pat_a \rangle = a \ \mathbf{in} \\
&\quad \mathbf{let} \ \langle pat_b \rangle = b \ \mathbf{in} \ e_2 \\
\mathbf{fun} \ (\langle pat_x \rangle : t) \rightarrow e &\Rightarrow \mathbf{fun} \ (x : t) \rightarrow \mathbf{let} \ \langle pat_x \rangle = x \ \mathbf{in} \ e \\
arg &::= \langle pat \rangle : t \mid 'x \text{ (fractional permission variable)} \\
\mathbf{fun} \ \langle arg_{1..n} \rangle \rightarrow e &\Rightarrow \mathbf{fun} \ \langle arg_1 \rangle \rightarrow .. \mathbf{fun} \ \langle arg_n \rangle \rightarrow e \\
\mathbf{let} \ f \ \langle arg_{1..n} \rangle = e_1 \ \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ f = \mathbf{fun} \ \langle arg_{1..n} \rangle \rightarrow e_1 \ \mathbf{in} \ e_2 \\
\mathbf{let} \ !f \ \langle arg_{1..n} \rangle = e_1 \ \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ \mathbf{Many} \ f = \mathbf{Many} \ (\mathbf{fun} \ \langle arg_{1..n} \rangle \rightarrow e_1) \ \mathbf{in} \ e_2 \\
\mathbf{fixpoint} &\equiv \mathbf{fix} \ (f, x : t, \mathbf{fun} \ \langle arg_{0..n} \rangle \rightarrow e_1 : t') \\
\mathbf{let} \ \mathbf{rec} \ f \ (x : t) \ \langle arg_{0..n} \rangle : t' = e_1 \ \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ f = \mathbf{fixpoint} \ \mathbf{in} \ e_2 \\
\mathbf{let} \ \mathbf{rec} \ !f \ (x : t) \ \langle arg_{0..n} \rangle : t' = e_1 \ \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ \mathbf{Many} \ f = \mathbf{Many} \ \mathbf{fixpoint} \ \mathbf{in} \ e_2
\end{aligned}$$

■ **Figure 2** Desugaring NUMLIN.

## 23:4 NumLin: Linear Types for Linear Algebra

87 and matrices [19]. Due to linearity, values cannot alias and are only used once, which means  
88 that linearly-typed updates result in no *observable* mutation.

89 As a result, programmers can reason about NUMLIN expressions as if they were ordinary  
90 mathematical expressions – as indeed they are! We are merely adopting a stricter type  
91 discipline than usual to make managing memory safe.

### 92 2.2.1 Intuitionism: ! and Many

93 However, linearity by itself is not sufficient to produce an expressive enough programming  
94 language. For values such as booleans, integers, floating-point numbers as well as pure  
95 functions, we need to be able to use them *intuitionistically*, that is, more than once or not at  
96 all. For this reason, we have the ! constructor at the type level and its corresponding **Many**  
97 constructor and **let Many <id> = .. in ..** eliminator at the term level. Because we want  
98 to restrict how a programmer can alias pointers and prevent a programmer from ignoring  
99 them (a memory leak), NUMLIN enforces simple syntactic restrictions on which values can  
100 be wrapped up in a **Many** constructor (details in Section 3).

### 101 2.2.2 Fractional Permissions

102 There are also valid cases in which we would want to alias pointers to a matrix. The most  
103 common is exemplified by the BLAS routine **gemm**, which (rather tersely) stands for *GEneric*  
104 *Matrix Multiplication*. A *simplified* definition of **gemm**( $\alpha$ , A, B,  $\beta$ , C) is  $C := \alpha AB + \beta C$ .  
105 In this case, A and B may alias each other but neither may alias C, because it is being written  
106 to. Related to *mutating* arrays and matrices is *freeing* them. Here, we would also wish to  
107 restrict aliasing so that we do not free one alias and then attempt to use another. Although  
108 linearity on its own suffices to prevent use-after-free errors when values are *not* aliased (a  
109 freed value is *out of scope* for the rest of the expression), we still need another simple, yet  
110 powerful concept to provide us with the extra expressivity of aliasing *without* losing any of  
111 the benefits of linearity.

112 Fractional permissions provide exactly this. Concretely, types of (pointers to) arrays  
113 and matrices are *parameterised* by a *fraction*. A fraction is either 1 ( $2^0$ ) or exactly *half* of  
114 another fraction ( $2^{-k}$ , for natural  $k$ ). The former represents complete ownership of that  
115 value: the programmer may mutate or free that value as they choose; the latter represents  
116 read-only access or a *borrow*: the programmer may read from the value but not write to or  
117 free it. Creating an array/matrix gives you ownership of it, so too does having one (with a  
118 fractional permission of  $2^0$ ) passed in as an argument.

119 In NUMLIN, we can produce two aliases of a single array/matrix, by *sharing* it. If the  
120 original alias had a fractional permission of  $2^{-k}$  then the two new aliases of it will have a  
121 fractional permission of  $2^{-(k+1)}$  each. Thanks to linearity, the original array/matrix with a  
122 fractional permission of  $2^{-k}$  will be out of scope after the sharing. When an array/matrix is  
123 shared as such, we can prevent the programmer from freeing or mutating it by making the  
124 types of **free** and **set** (for mutation) require a *whole* ( $2^0$ ) permission.

125 If we have two aliases *to the same matrix* with *identical* fractional permissions ( $2^{-(k+1)}$ ),  
126 we can recombine or *unshare* them back into a single one, with a larger  $2^{-k}$  permission. As  
127 before, thanks to linearity, the original two aliases will be out of scope after unsharing.

### 128 2.2.3 Runtime Errors

129 Aside from out-of-bounds indexing, matrix unsharing is one of only *two* operations that  
130 can fail at runtime (the other being dimension checks, such as for **gemm**). The check being

```

let rec factorial ( !x : !int ) : !int =
  if x < 0 || x = 0 then
    1
  else
    x * factorial (x - 1) in
  factorial ;;

```

■ **Figure 3** Factorial function in NUMLIN.

131 performed is a simple sanity check that the two aliasing pointers passed to `unshare` point to  
 132 the same array/matrix. Section 5 contains an overview of how we could remove the need for  
 133 this by tracking pointer identities statically by augmenting the type system further.

## 134 2.2.4 Recursion

135 The final feature of NUMLIN which makes it sufficiently expressive is recursion (and of  
 136 course, conditional branches to ensure termination). Conditional branches are implemented  
 137 by ensuring that both branches use the same set of linear values. A function can be recursive  
 138 if it captures no linear values from its environment. Like with `Many`, this is enforced via  
 139 simple syntactic restrictions on the definition of recursive functions.

## 140 2.3 Examples

### 141 2.3.1 Factorial

142 Although a factorial function (Figure 3) may seem like an aggressively pedestrian first example,  
 143 in a linearly typed language such as NUMLIN it represents the culmination of many features.

144 To simplify the design and implementation of NUMLIN’s type system, recursive functions  
 145 must have full type annotations (non-recursive functions need only their argument types  
 146 annotated). Its body is a closed expression (with respect to the function’s arguments), so it  
 147 type-checks (since it does not capture any linear values from its environment).

148 The only argument is `!x : !int`. The `!` annotation on `x` is a syntactic convenience for  
 149 declaring the value to be used intuitionistically, its full and precise meaning is described in  
 150 Section 3.1.

151 The condition for an `if` may or may not use linear values (here, with `x < 0 || x = 0`, it  
 152 does not). Any linear values used by the condition would not be in scope in either branch of  
 153 the `if`-expression. Both branches use `x` differently: one ignores it completely and the other  
 154 uses it twice.

155 All numeric and boolean literals are implicitly wrapped in a `Many` and all primitives  
 156 involving them return a `!int`, `!bool` or `!elt` (types of elements of arrays/matrices, typically  
 157 64-bit floating-point numbers). The short-circuiting `||` behaves in exactly the same way as a  
 158 boolean-valued `if`-expression.

### 159 2.3.2 Summing over an Array

160 Now we can add fractional permissions to the mix: Figure 4 shows a simple, tail-recursive  
 161 implementation of summing all the elements in an array. There are many new features; first  
 162 among them is `!x0 : !elt`, the type of array/matrix elements (64-bit floating point).

163 Second is `(!x) (row: !x arr)` which is an array with a universally-quantified fractional  
 164 permission. In particular, this means the body of the function cannot mutate or free the

## 23:6 NumLin: Linear Types for Linear Algebra

```

let rec sum_array (!i : !int) (!n : !int) (!x0 : !elt)
  ('x) (row : 'x arr) : 'x arr * !elt =
  if i = n then
    (row, x0)
  else
    let (row, !x1) = row[i] in
    sum_array (i + 1) n (x0 +. x1) 'x row in
  sum_array ;;

```

■ **Figure 4** Summing over an array in NUMLIN.

```

let row = row[i] := x1 in (* or *) let () = free row in
(* Could not show equality: *)
(*      z arr *)
(* with *)
(*      'x arr *)
(* *)
(* Var 'x is universally quantified *)
(* Are you trying to write to/free/unshare an array you don't own? *)
(* In examples/sum_array.lt, at line: 7 and column: 19 *)

```

■ **Figure 5** Attempting to write to or free a read only array in NUMLIN.

165 input array, only read from it. If the programmer did try to mutate or free `row`, then they  
 166 would get a helpful error message (Figure 5).

167 Alongside taking a `row : 'x arr`, the function also returns an array with exactly the  
 168 same fractional permission as the `row` (which can only be `row`). This is necessary because of  
 169 linearity: for the caller, the original array passed in as an argument would be out of scope  
 170 for the rest of the expression, so it needs to be returned and then rebound to be used for the  
 171 rest of the function.

172 An example of this consuming and re-binding is in `let (row, !x1) = row[i]`. Indexing  
 173 is implemented as a primitive `get : 'x. 'x arr  $\multimap$  !int  $\multimap$  'x arr  $\otimes$  !elt`. Although fractional  
 174 permissions can be passed around explicitly (as done in the recursive call), they can also  
 175 be *automatically inferred at call sites*: `row[i] == get _ row i` takes advantage of this  
 176 convenience.

### 177 2.3.3 One-dimensional Convolution

178 Figure 6 extends the set of features demonstrated by the previous examples by mutating one  
 179 of the input arrays. A one-dimensional convolution involves two arrays: a read-only kernel  
 180 (array of weights) and an input vector. It modifies the input vector *in-place* by replacing  
 181 each `write[i]` with a weighted (as per the values in the kernel) sum of it and its neighbours;  
 182 intuitively, sliding a dot-product with the kernel across the vector.

183 What's implemented in Figure 6 is a *simplified* version of this idea, so as to not distract  
 184 from the features of NUMLIN. The simplifications are:

- 185 ■ the kernel has a length 3, so only the value of `write[i-1]` (prior to modification in the  
 186 previous iteration) needs to be carried forward using `x0`
- 187 ■ `write` is assumed to have length `n+1`
- 188 ■ `i`'s initial value is assumed to be 1
- 189 ■ `x0`'s initial value is assumed to be `write[0]`
- 190 ■ the first and last values of `write` are ignored.

```

let rec simp_oned_conv
  (!i : !int) (!n : !int) (!x0 : !elt)
  (write : z arr) ('x) (weights : 'x arr)
  : 'x arr * z arr =
  if n = i then (weights, write) else
  let !w0 <- weights[0] in
  let !w1 <- weights[1] in
  let !w2 <- weights[2] in
  let !x1 <- write[i] in
  let !x2 <- write[i + 1] in
  let written = write[i] := w0 *. x0 +. (w1 *. x1 +. w2 *. x2) in
  simp_oned_conv (i + 1) n x1 written _ weights in
simp_oned_conv ;;

```

■ Figure 6 *Simplified* one-dimensional convolution.

```

symm  : !bool → !elt → 'x. 'x mat → 'y. 'y mat → !elt → z mat →
        ('x mat ⊗ 'y mat) ⊗ z mat
gemm  : !elt → 'x. 'x mat ⊗ !bool → 'y. 'y mat ⊗ !bool → !elt →
        z mat → ('x mat ⊗ 'y mat) ⊗ z mat
gesv  : z mat → z mat → z mat ⊗ z mat
posv  : z mat → z mat → z mat ⊗ z mat
potrs : 'x. 'x mat → z mat → 'x mat ⊗ z mat
syrrk : !bool → !elt → 'x. 'x mat → !elt → z mat → 'x mat ⊗ z mat

```

■ Figure 7 Types of some NUMLIN primitives.

191 Mutating an array is implemented similarly to indexing one – a primitive `set : z arr →`  
 192 `!int → !elt → z arr`. It consumes the original array and returns a new array with the  
 193 updated value. `let written = write[i] := <exp>` is just syntactic sugar for `let written`  
 194 `= set write i <exp>`.

195 Since `write : z arr` (where `z` stands for  $k = 0$ , representing a fractional permission of  
 196  $2^{-k} = 2^{-0} = 1$ ), we may mutate it, but since we only need to read from `weights`, its  
 197 fractional permission index can be universally-quantified. In the recursive call, we see `_` being  
 198 used explicitly to tell the compiler to *infer* the correct fractional permission based on the  
 199 given arguments.

### 200 2.3.4 Digression: Types of Primitives

201 *The most pertinent aspect of NUMLIN is the types of its primitives* (Figure 7). While the types  
 202 of operations such as `get` and `set` might be borderline obvious, the types of BLAS/LAPACK  
 203 routines become an *incredibly useful, automated check for using the API correctly*.

204 We determine the types for these routines by consulting their documentation. Each  
 205 routine has a record of the expected aliasing behaviour and whether or not it modifies or  
 206 consumes its argument in any way. We use that to derive the types in Figure 7. Since  
 207 most of these low-level routines are very careful not to do any allocation themselves, it is  
 208 generally very easy to give each a NUMLIN type – every argument that can modify/consume

```

let !square ('x) (x : 'x mat) =
  let (x, (!m, !n)) = sizeM _ x in
  let (x1, x2) = shareM _ x in
  let answer <- new (m, n) [| x1 * x2 |] in
  let x = unshareM _ x1 x2 in
  (x, answer) in
  square
;;

```

■ **Figure 8** Linear regression (OLS):  $\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$

its argument needs a full permission, and all others can be fraction-polymorphic. Taking Fortran as an example, it has a notion of `in`, `out` and `inout` parameters. The latter two would need full `z` permissions; the first would be fraction-polymorphic.

### 2.3.5 Squaring a Matrix

Figure 8 shows how a linearly-typed matrix squaring function may be written in NUMLIN. It is a *non-recursive* function declaration (the return type is inferred). Since we would like to be able to use a function like `square` more than once, it is marked with a `!` annotation (which also ensures it captures no linear values from the surrounding environment).

To square a matrix, first, we extract the dimensions of the argument `x`. Then, because we need to use `x` twice (so that we can multiply it by itself) but linearity only allows one use, we use `shareM`:  $'x. 'x \text{ mat} \multimap 'x \text{ s mat} \otimes 'x \text{ s mat}$  to split the permission `'x` (which represents  $2^{-x}$ ) into two halves (`'x s`, which represents  $2^{-(x+1)}$ ).

Even if `x` had type `z mat`, sharing it now enforces the assumption of all BLAS/LAPACK routines that any matrix which is written to (which, in NUMLIN, is always of type `z mat`) does not alias any other matrix in scope. So if we did try to use one of the aliases in mutating way, the expression would not type check, and we would get an error similar to the one in Figure 5.

The line `let answer <- new (m,n) [| x1 * x2 |]` is syntactic sugar for first creating a new  $m \times n$  matrix (`let answer = matrix m n`) and then storing the result of the multiplication in it (`let ((x1, x2), answer) = gemm 1. _ (x1, false) _ (x2, false) 0. answer`). `false` means the matrix should not be accessed with indices transposed.

By using some simple pattern-matching and syntactic sugar (Figure 9), we can:

- write normal-looking, apparently non-linear code
- use matrix expressions directly and have a call to an efficient call to a BLAS/LAPACK routine inserted with appropriate re-bindings
- retain the safety of linear types with fractional permissions by having the compiler statically enforce the aliasing and read/write rules implicitly assumed by BLAS/LAPACK routines.

### 2.3.6 Linear Regression

In Figure 10, we wish to compute  $\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$ . To do that, first, we extract the dimensions of matrix `x`. Then, we say we would like `xy` to be a new matrix, of dimension  $m \times 1$ , which contains the result of  $\mathbf{X}^T \mathbf{y}$  (using syntactic sugar for `matrix` and `gemm` calls similar to that used in Figure 8, with a `^T` annotation on `x` to set `x`'s 'transpose indices'-flag to `true`).



$$\begin{aligned} \text{let } v \leftarrow x[e] \text{ in } e &\Rightarrow \text{let } (x, !v) = x[e] \text{ in } e && \text{(similarly for matrices)} \\ \text{let } x_2 \leftarrow \text{new } [| x_1 |] \text{ in } e &\Rightarrow \text{let } (x_1, x_2) = \text{copyM\_} x_1 \text{ in } e \\ \text{let } x_2 \leftarrow [| x_1 |] \text{ in } e &\Rightarrow \text{let } (x_1, x_2) = \text{copyM\_to\_} x_1 \ x_2 \text{ in } e \\ M ::= X \mid X^T \mid \text{sym}(X) \\ \\ \text{let } Y \leftarrow \text{new } (n, k) [| \alpha M_1 M_2 |] \text{ in } e &\Rightarrow \\ \text{let } Y = \text{matrix } n \ k \text{ in let } Y \leftarrow [| \alpha M_1 M_2 + 0Y |] \text{ in } e \\ \text{let } Y \leftarrow [| \alpha X X^T + \beta Y |] \text{ in } e &\Rightarrow \\ \text{let } (X, Y) = \text{syrk false } \alpha \_ X \ \beta \ Y \text{ in } e \\ \text{let } Y \leftarrow [| \alpha X^T X + \beta Y |] \text{ in } e &\Rightarrow \\ \text{let } (X, Y) = \text{syrk true } \alpha \_ X \ \beta \ Y \text{ in } e \\ \text{let } Y \leftarrow [| \alpha \text{sym}(X_1) X_2 + \beta Y |] \text{ in } e &\Rightarrow \\ \text{let } ((X_1, X_2), Y) = \text{symm false } \alpha \_ X_1 \_ X_2 \ \beta \ Y \text{ in } e \\ \text{let } Y \leftarrow [| \alpha X_2 \text{sym}(X_1) + \beta Y |] \text{ in } e &\Rightarrow \\ \text{let } ((X_1, X_2), Y) = \text{symm true } \alpha \_ X_1 \_ X_2 \ \beta \ Y \text{ in } e \\ \text{let } Y \leftarrow [| \alpha X_1^{T?} X_2^{T?} + \beta Y |] \text{ in } e &\Rightarrow \\ \text{let } ((X_1, X_2), Y) = \text{gemm } \alpha \_ (X_1, \text{true}_{\text{false}}) \_ (X_2, \text{true}_{\text{false}}) \ \beta \ Y \text{ in } e \end{aligned}$$

■ **Figure 9** Purely syntactic pattern-matching translations of matrix expressions.

243 However, the line `let x_T_x <- new (m,m) [| x^T * x |]`, works for a slightly differ-  
 244 ent reason: that pattern is matched to a BLAS call to `(syrk true 1. x 0. x_T_x)`, which  
 245 only uses `x` once. Hence `x` can appear *twice* in the *pattern* without any calls to `share`.

246 After computing `x_T_x`, we need to invert it and then multiply it by `xy`. The BLAS  
 247 routine `posv : z mat  $\rightarrow$  z mat  $\rightarrow$  z mat  $\otimes$  z mat` does exactly that: assuming the first  
 248 argument is symmetric, `posv` mutates its second argument to contain the desired value. Its  
 249 first argument is also mutated to contain the (upper triangular) Cholesky decomposition  
 250 factor of the original matrix. Since we do not need that matrix (or its memory) again, we  
 251 `free` it. If we forgot to, we would get a `Variable to_del not used` error. Lastly, we return  
 252 the `answer` alongside the untouched input matrices `(x,y)`.

### 253 2.3.7 L1-norm Minimisation on Manifolds

254 L1-norm minimisation is often used in optimisation problems, as a *regularisation* term for  
 255 reducing the influence of outliers. Although the below formulation [11] is intended to be used  
 256 with *sparse* computations, NUMLIN's current implementation only implements dense ones.  
 257 However, it still serves as a useful example of explaining NUMLIN's features.

258 Figure 11 shows even more pattern-matching. Patterns of the form `let <id> <- [|`  
 259 `beta * c + alpha * a * b |]` are also desugared to `gemm` calls. Primitives like `transpose :`  
 260 `'x. 'x mat  $\rightarrow$  'x mat  $\otimes$  z mat` and `eye : 'int  $\rightarrow$  z mat` allocate new matrices; `transpose`  
 261 returns the transpose of a given matrix and `eye k` evaluates to a  $k \times k$  identity matrix.

262 We also see our first example of re-using memory for different matrices: like with `to_del`

```

let !lin_reg ('x) (x : 'x mat)
    ('y) (y : 'y mat) =
  let (x, (!_n, !m)) = sizeM _ x in
  let xy <- new (m, 1) [| xT * y |] in
  let x_T_x <- new (m, m) [| xT * x |] in
  let (to_del, answer) = posv x_T_x xy in
  let () = freeM to_del in
  ((x, y), answer) in
lin_reg ;;

```

■ Figure 10 Linear regression (OLS):  $\hat{\beta} = (X^T X)^{-1} X^T y$

```

let !l1_norm_min (q : z mat) (u : z mat) =
  let (u, (!_n, !k)) = sizeM _ u in
  let (u, u_T) = transpose _ u in
  let (tmp_n_n, q_inv_u) = gesv q u in
  let i = eye k in
  let to_inv <- [| i + u_T * q_inv_u |] in
  let (tmp_k_k, inv_u_T) = gesv to_inv u_T in
  let () = freeM tmp_k_k in
  let answer <- [| 0. * tmp_n_n + q_inv_u * inv_u_T |] in
  let () = freeM q_inv_u in
  let () = freeM inv_u_T in
  answer in
l1_norm_min ;;

```

■ Figure 11 L1-norm minimisation on manifolds:  $Q^{-1}U(I + U^T Q^{-1}U)^{-1}U^T$

263 and `posv` in the previous example, we do not need the value stored in `tmp_5_5` after the call  
 264 to `gesv` (a primitive similar to `posv` but for a non-symmetric first argument). However, we  
 265 can re-use its memory much later to store `answer` with `let answer <- [| 0. * tmp_5_5`  
 266 `+ q_inv_u * inv_u_T |]`. Again, thanks to linearity, the identifiers `q` and `tmp_5_5` are out  
 267 of scope by the time `answer` is bound. Although during execution, all three refer to the same  
 268 piece of memory, logically they represent different values throughout the computation.

### 269 2.3.8 Kalman Filter

270 A *Kalman Filter* [15] is an algorithm for combining prior knowledge of a state, a statistical  
 271 model and measurements from (noisy) sensors to produce an estimate a more reliable  
 272 estimated of the current state. It has various applications (navigation, signal-processing,  
 273 econometrics) and is relevant here because it is usually presented as a series of complex  
 274 matrix equations.

275 Figure 12 shows a NUMLIN implementation of a Kalman filter (equations in Figure 13).  
 276 A few new features and techniques are used in this implementation:

- 277 ■ `sym` annotations in matrix expressions: when this is used, a call to `symm` (the equivalent  
 278 of `gemm` but for symmetric matrices so that only half the operations are performed) is  
 279 inserted
- 280 ■ `copyM_to` is used to re-use memory by *overwriting* the contents of its second argument  
 281 to that of its first (erroring if dimensions do not match)
- 282 ■ `let new_r <- new [| r_2 |]` creates a copy of `r_2`
- 283 ■ `posvFlip` is like `posv` except for solving  $XA = B$
- 284 ■ a lot of memory re-use; the following sets of identifiers alias each other:

```

let !kalman
  ('s) (sigma : 's mat) (* n,n *)
  ('h) (h : 'h mat)      (* k,n *)
  (mu : z mat)            (* n,1 *)
  (r_1 : z mat)           (* k,k *)
  (data_1 : z mat)        (* k,1 *) =
  let (h, (!k, !n)) = sizeM _ h in
  (* could use [| sym(sigma) * hT |] but would
     need a (n,k) temporary hT = transpose _ h *)
  let sigma_hT <- new (n, k) [| sigma * h^T |] in
  let r_2 <- [| r_1 + h * sigma_hT |] in
  let (k_by_k, x) = posvFlip r_2 sigma_hT in
  let data_2 <- [| h * mu - data_1 |] in
  let new_mu <- [| mu + x * data_2 |] in
  let x_h <- new (n,n) [| x * h |] in
  let () = freeM (* n,k *) x in
  let sigma2 <- new [| sigma |] in
  let new_sigma <- [| sigma2 - x_h * sym(sigma) |] in
  let () = freeM (* n,n *) x_h in
  ((sigma, h), (new_sigma, (new_mu, (k_by_k, data_2)))) in
kalman ;;

```

■ **Figure 12** Kalman filter: see Figure 13 for the equations this code implements and Figure 20 for an equivalent CBLAS/LAPACKE implementation.

$$\mu' = \mu + \Sigma H^T (R + H \Sigma H^T)^{-1} (H \mu - \text{data})$$

$$\Sigma' = \Sigma (I - H^T (R + H \Sigma H^T)^{-1} H \Sigma)$$

■ **Figure 13** Kalman filter equations (credit: matthewrocklin.com).

285    — r\_1, r\_2 and k\_by\_k  
 286    — data\_1 and data\_2  
 287    — mu and new\_mu  
 288    — sigma\_hT and x

289    The NUMLIN implementation is much longer than the mathematical equations for two  
 290    reasons. First, the NUMLIN implementation is a let-normalised form of the Kalman equations:  
 291    since there a large number of unary/binary (and occasionally ternary) sub-expressions in  
 292    the equations, naming each one line at a time makes the implementation much longer.  
 293    Second, NUMLIN has the additional task of handling explicit allocations, aliasing and frees  
 294    of matrices. However, it is exactly this which makes it possible (and often, easy) to spot  
 295    additional opportunities for memory re-use. Furthermore, a programmer can explore those  
 296    opportunities easily because NUMLIN's type system statically enforces correct memory  
 297    management and the aliasing assumptions of BLAS/LAPACK routines.

## 298 3 Formal System

### 299 3.1 Core Type Theory

300    The full typing rules are in Appendix A.1, but the key ideas are as follow.

301    ■ A typing judgement consists of  $\Theta; \Delta; \Gamma \vdash e : t$ .

## 23:12 NumLin: Linear Types for Linear Algebra

- 302 ■  $\Theta$  is the environment that tracks which fractional permission variables in scope. Fractional
- 303 permissions (the `Perm` judgement) and types (the `Type` judgement) are *well-formed* if all
- 304 of their free fractional variables are in  $\Theta$ .
- 305 ■  $\Delta$  is the environment storing non-linearly or *intuitionistically* typed variables.
- 306 ■  $\Gamma$  is the environment storing linearly typed variables.

307 Note that rules for typing `()`, booleans, integers and elements are typed with respect to  
 308 an *empty* linear environment: this means no linear values are needed to produce a value of  
 309 those types.

$$310 \quad \frac{}{\Theta; \Delta; \cdot \vdash () : \mathbf{unit}} \text{TY\_UNIT\_INTRO}$$

311 Conversely, whenever two or more subexpressions need to be typed, they must consume  
 312 a disjoint set of linear values (pairs, let-expressions). In the case of if-expressions, both  
 313 branches must consume the same set of linear values (disjoint to the ones used to evaluate  
 314 the condition).

$$315 \quad \frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : !\mathbf{bool} \\ \Theta; \Delta; \Gamma' \vdash e_1 : t' \\ \Theta; \Delta; \Gamma' \vdash e_2 : t' \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{if } e \mathbf{ then } e_1 \mathbf{ else } e_2 : t} \text{TY\_BOOL\_ELIM}$$

316 The **Many** introduction and elimination rules are very important. Producing `!`-type values  
 317 may only be done if the expression inside is a syntactic value which is not a location. This  
 318 allows all safely duplicable resources, including functions which capture non-linear resources  
 319 from their environments, but prevents producing aliases of (pointers to) arrays and matrices.  
 320 This is exactly the same as value-restriction from the world of parametric polymorphism;  
 321 without it, the expression `let Many x = Many (array 5) in let () = free x in x[0]` would  
 322 type-check but error at runtime.

$$323 \quad \frac{\begin{array}{l} \Theta; \Delta; \cdot \vdash v : t \\ v \neq l \end{array}}{\Theta; \Delta; \cdot \vdash \mathbf{Many } v : !t} \text{TY\_BANG\_INTRO}$$

324 Consuming a variable that refers to a `!`-type value *moves it* from the linear environment  
 325  $\Gamma$  and *into* the intuitionistic environment  $\Delta$ .

$$326 \quad \frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : !t \\ \Theta; \Delta, x : t; \Gamma' \vdash e' : t' \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{let Many } x = e \mathbf{ in } e' : t'} \text{TY\_BANG\_ELIM}$$

327 Using this, we can explain how the `!` annotation on variables – first introduced in the  
 328 factorial example in 2.3.1 – works. That is, we can explain why the meaning of `let !x = e in e'`  
 329 can be expressed using only the rules presented thus far, as `let Many x = e in let Many x =`  
 330 `Many (Many x) in e'`.<sup>1</sup> The reader is invited to quickly convince themselves that the

<sup>1</sup> Why we have this at all is for the sake of ergonomics when using binary arithmetic operations (e.g. of type `!int ⊗ !int → !int`): a programmer should be able to write `let x = 5 + 5 in x - x`, which, although non-linear in `x`, is morally right because integers and operations on them rarely need to be linear. Though it should be possible to handle this using a LNL-style presentation of linear types [5] (using adjoint modalities to distinguish between intrinsically linear and intrinsically intuitionistic types) that is a pretty big digression from the stated goals of this paper.

331 following meta-rule is provable using `TY_BANG_INTRO` (twice), `TY_BANG_ELIM` (twice)  
 332 and weakening the intuitionistic environment  $\Delta$  (once).

$$333 \quad \frac{\Theta; \Delta; \Gamma \vdash e : !t \quad \Theta; \Delta, x : !t; \Gamma' \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash \text{let } !x = e \text{ in } e' : t'} \quad \text{META\_TY\_LET\_BANG}$$

334 Rules `TY_GEN` and `TY_SPC` are for fractional permission generalisation and specialisa-  
 335 tion respectively. They allow the definition and use of functions that are polymorphic in the  
 336 fractional permission index of their results and one or more of their arguments.

$$337 \quad \frac{\Theta, fc; \Delta; \Gamma \vdash e : t}{\Theta; \Delta; \Gamma \vdash \text{fun } 'fc \rightarrow e : 'fc.t} \quad \text{TY\_GEN} \qquad \frac{\Theta \vdash f \text{ Perm} \quad \Theta; \Delta; \Gamma \vdash e : 'fc.t}{\Theta; \Delta; \Gamma \vdash e[f] : t[f/fc]} \quad \text{TY\_SPC}$$

338 Rule `TY_FIX` shows how recursive functions are typed. Even though recursive functions  
 339 are fully annotated, type checking them is interesting for two reasons: to type check the  
 340 body of the fixpoint, the type of the recursive function is in the *intuitionistic* environment  $\Delta$   
 341 (without this, you would not be able to write a base case) whilst the argument and its type  
 342 are the *only things in the linear environment*  $\Gamma$ . The latter means that recursive functions  
 343 can be type checked in an empty environment (thus be wrapped in `Many` and used zero or  
 344 multiple times).

$$345 \quad \frac{\Theta; \Delta, g : t \multimap t'; \cdot, x : t \vdash e : t'}{\Theta; \Delta; \cdot \vdash \text{fix } (g, x : t, e : t') : t \multimap t'} \quad \text{TY\_FIX}$$

346 Lastly, types of almost all `NUMLIN` primitives, as embedded in OCaml's type system, are  
 347 shown in Appendix F, with some similar ones (like those for binary arithmetic operators)  
 348 omitted for brevity. The main difference between the OCaml type of a primitive like `gemm`  
 349 and its `NUMLIN` counterpart (Figure 7) is the inclusion of explicit universal-quantification of  
 350 fractional permission variables in the latter.

## 351 3.2 Dynamic Semantics

352 The full, small-step transition relation is in Appendix A.2, but the key ideas are as follow.

353 Heaps ( $\sigma$ ) are multisets containing triples of an abstract location  $l$ , a fractional permission  
 354  $f$  and sized matrices  $m_{n,k}$ . The notation  $l \mapsto_f m_{k_1, k_2}$  should be read as “location  $l$  represents  
 355  $f$  ownership over matrix  $m$  (of size  $k_1 \times k_2$ )”. Each heap-and-expression either steps to  
 356 another heap-and-expression or a runtime error `err`. In the full grammar definition we see a  
 357 definition of values and contexts in the language.

358 We draw the reader's attention to the definitions relating to fractional permissions.  
 359 Specifically, unlike a lambda, the body of a `fun`  $fc \rightarrow v$  must be a syntactic value. The  
 360 context `fun`  $fc \rightarrow [-]$  means expressions can be reduced inside a fractional permission  
 361 generalisation. This is to emphasize that fractions are merely *compile-time constructs* and  
 362 do not affect runtime behaviour. Correct usage of fractions is enforced by the type system,  
 363 so programs do not get stuck. Fractional permissions are specialised using substitution over  
 364 both the heap and an expression (`OP_FRAC_PERM`).

$$365 \quad \overline{\langle \sigma, (\text{fun } 'fc \rightarrow v)[f] \rangle \rightarrow \langle \sigma[f/fc], v[f/fc] \rangle} \quad \text{OP\_FRAC\_PERM}$$

366 Like with the static semantics, the interesting rules in the dynamic semantics are those  
 367 relating to primitives. Creating a matrix (`matrix`  $k_1 \ k_2$ ) successfully (`OP_MATRIX`) requires

## 23:14 NumLin: Linear Types for Linear Algebra

non-negative dimensions and returns a (fresh) location of a matrix of those dimensions, extending the heap to reflect that  $l$  represents a complete ownership over the new matrix.

$$\frac{0 \leq k_1, k_2 \quad l \text{ fresh}}{\langle \sigma, \mathbf{matrix} \ k_1 \ k_2 \rangle \rightarrow \langle \sigma + \{l \mapsto_1 M_{k_1, k_2}\}, l \rangle} \quad \text{OP\_MATRIX}$$

Dually, OP\_FREE, requires a location represent complete ownership before removing it and the matrix it points to from the heap.

$$\frac{}{\langle \sigma + \{l \mapsto_1 m_{k_1, k_2}\}, \mathbf{free} \ l \rangle \rightarrow \langle \sigma, () \rangle} \quad \text{OP\_FREE}$$

Choosing a multiset representation as opposed to a set allows for two convenient invariants: multiplicity of a triple  $l \mapsto_f m_{k_1, k_2}$  in the heap corresponds to the number of aliases of  $l$  in the expression with type  $f$  **mat** and the sum of all the fractions for  $l$  will always be 1 (for a closed, well-typed expression). With this in mind, the rules OP\_SHARE and OP\_UNSHARE\_EQ are fairly natural.

$$\frac{}{\langle \sigma + \{l \mapsto_f m_{k_1, k_2}\}, \mathbf{share}[f] \ l \rangle \rightarrow \langle \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\}, (l, l) \rangle} \quad \text{OP\_SHARE}$$

$$\frac{\sigma' \equiv \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\}}{\langle \sigma', \mathbf{unshare}[f] \ l \ l \rangle \rightarrow \langle \sigma + \{l \mapsto_f m_{k_1, k_2}\}, l \rangle} \quad \text{OP\_UNSHARE\_EQ}$$

Combining all of these features, we see that OP\_GEMM\_MATCH requires that the location being updated ( $l_3$ ) has complete ownership of over matrix  $m_3$  and can thus change what value it stores to  $m_1 m_2 + m_3$ . In particular, this places no restriction on  $l_2$  and  $l_3$ : they could be **shared** aliases of the same matrix. Transition rules for other primitives (omitted) follow the same structure:  $\mapsto_1$  for any locations that are written to and  $\mapsto_{fc}$  for anything else.

$$\frac{\begin{aligned} \sigma' &\equiv \sigma + \{l_1 \mapsto_{fc_1} m_{1 \ k_1, k_2}\} + \{l_2 \mapsto_{fc_2} m_{2 \ k_2, k_3}\} \\ \sigma_1 &\equiv \sigma' + \{l_3 \mapsto_1 m_{3 \ k_1, k_3}\} \\ \sigma_2 &\equiv \sigma' + \{l_3 \mapsto_1 (m_1 \ m_2 + m_3)_{k_1, k_3}\} \end{aligned}}{\langle \sigma_1, \mathbf{gemm}[fc_1] \ l_1 [fc_2] \ l_2 \ l_3 \rangle \rightarrow \langle \sigma_2, ((l_1, l_2), l_3) \rangle} \quad \text{OP\_GEMM\_MATCH}$$

### 3.3 Logical Relation

First, we define an interpretation of heaps with fractional permissions in the style of Bornat et. al [9] (interpreting the multiset as a partial map from locations to the sum of all its associated fractions and a matrix) as well as the n-fold iteration of  $\rightarrow$ .

$$\mathcal{H}[\sigma] = \star_{(l, f, m) \in \sigma} [l \mapsto_f m]$$

where

$$(\varsigma_1 \star \varsigma_2)(l) \equiv \begin{cases} \varsigma_1(l) & \text{if } l \in \text{dom}(\varsigma_1) \wedge l \notin \text{dom}(\varsigma_2) \\ \varsigma_2(l) & \text{if } l \in \text{dom}(\varsigma_2) \wedge l \notin \text{dom}(\varsigma_1) \\ (f_1 + f_2, m) & \text{if } (f_1, m) = \varsigma_1(l) \wedge (f_2, m) = \varsigma_2(l) \wedge f_1 + f_2 \leq 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

We then define a step-indexed logical relation in the style of Morrisett et. al [2].  $(\varsigma, v) \in \mathcal{V}_k[t]$  means it takes a heap with exactly  $\varsigma$  resources to produce a value  $v$  of type  $t$  in at

most  $k$  steps. So, something like a **unit** or a  $!t$  need no resources, whereas a  $f$  **mat** needs exactly  $f$  ownership of a some matrix and a pair needs a  $\star$  combination of the heaps required for each component.

$$\begin{aligned} \mathcal{V}_k[\mathbf{unit}] &= \{(\emptyset, *)\} \\ \mathcal{V}_k[f \mathbf{mat}] &= \{(\{l \mapsto_{2-f} \_ \}, l)\} \\ \mathcal{V}_k[!t] &= \{(\emptyset, \mathbf{Many} \ v) \mid (\emptyset, v) \in \mathcal{V}_k[t]\} \\ \mathcal{V}_k[t_1 \otimes t_2] &= \{(\varsigma_1 \star \varsigma_2, (v_1, v_2)) \mid (\varsigma_1, v_1) \in \mathcal{V}_k[t_1] \wedge (\varsigma_2, v_2) \in \mathcal{V}_k[t_2]\} \end{aligned}$$

The definition of  $\mathcal{V}_k[fc. t]$  says a value and heap must be the same regardless of what fraction is substituted into both; the  $k - 1$  is to take into account fraction specialisation takes ones step (OP\_SPC).

$$\mathcal{V}_k[fc. t] = \{(\varsigma, \mathbf{fun} \ fc \rightarrow v) \mid \forall f. (\varsigma[f/fc], v[f/fc]) \in \mathcal{V}_{k-1}[t[f/fc]]\}$$

To understand the definition of  $\mathcal{V}_k[t' \multimap t]$ , we must first look at  $\mathcal{C}_k[t]$ , the computational interpretation of types. Intuitively, it is a combination of a frame rule on heaps (no interference), type-preservation and termination (in  $j < k$  steps) to either an error or a heap-and-expression, with the further condition that if the expression is a syntactic value then it is also one semantically.

$$\begin{aligned} \mathcal{C}_k[t] &= \{(\varsigma_s, e_s) \mid \forall j < k, \sigma_r. \varsigma_s \star \sigma_r \text{ defined} \Rightarrow \langle \sigma_s + \sigma_r, e_s \rangle \rightarrow^j \mathbf{err} \vee \exists \sigma_f, e_f. \\ &\quad \langle \sigma_s + \sigma_r, e_s \rangle \rightarrow^j \langle \sigma_f + \sigma_r, e_f \rangle \wedge (e_f \text{ is a value} \Rightarrow (\varsigma_f \star \sigma_r, e_f) \in \mathcal{V}_{k-j}[t])\} \end{aligned}$$

In this light,  $\mathcal{V}_k[t' \multimap t]$  simply says that  $v$  is a function and that evaluating the application of it to any argument (of the correct type, requiring its own set of resources, bounded by  $k$  steps) satisfies all the aforementioned properties.

$$\begin{aligned} \mathcal{V}_k[t' \multimap t] &= \{(\varsigma_v, v) \mid (v \equiv \mathbf{fun} \ x : t' \rightarrow e \vee v \equiv \mathbf{fix}(g, x : t', e : t)) \wedge \\ &\quad \forall j \leq k, (\varsigma_{v'}, v') \in \mathcal{V}_j[t']. \varsigma_v \star \varsigma_{v'}' \text{ defined} \Rightarrow (\varsigma_v \star \varsigma_{v'}', v v') \in \mathcal{C}_j[t]\} \end{aligned}$$

The interpretation of typing environments  $\Delta$  and  $\Gamma$  are with respect to an arbitrary substitution of fractional permissions  $\theta$ . Note that only the interpretation of  $\Gamma$  involves a (potentially) non-empty heap.

$$\begin{aligned} \mathcal{I}_k[\Delta, x : t]\theta &= \{\delta[x \mapsto v_x] \mid \delta \in \mathcal{I}_k[\Delta]\theta \wedge (\emptyset, v_x) \in \mathcal{V}_k[\theta(t)]\} \\ \mathcal{L}_k[\Gamma, x : t]\theta &= \{(\varsigma \star \varsigma_x, \gamma[x \mapsto v_x]) \mid (\varsigma, \gamma) \in \mathcal{L}_k[\Gamma]\theta \wedge (\varsigma_x, v_x) \in \mathcal{V}_k[\theta(t)]\} \end{aligned}$$

And so the final semantic interpretation of a typing judgement simply quantifies over all possible fractional permission substitutions  $\theta$ , linear value substitutions  $\gamma$ , intuitionistic value substitutions  $\delta$  and heaps  $\sigma$ . Note that,  $\varsigma \equiv \mathcal{H}[\theta(\sigma)]$ .

$$\begin{aligned} {}_k[\Theta; \Delta; \Gamma \vdash e : t] &= \forall \theta, \delta, \gamma, \sigma. \Theta = \text{dom}(\theta) \wedge (\varsigma, \gamma) \in \mathcal{L}_k[\Gamma]\theta \wedge \delta \in \mathcal{I}_k[\Delta]\theta \Rightarrow \\ &\quad (\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\theta(t)] \end{aligned}$$

### 3.4 Soundness Theorem

► **Theorem 1.** (*The Fundamental Lemma of Logical Relations*)

$$\forall \Theta, \Delta, \Gamma, e, t. \Theta; \Delta; \Gamma \vdash e : t \Rightarrow \forall k. {}_k[\Theta; \Delta; \Gamma \vdash e : t]$$

### 438 3.4.1 Explanation

439 If an expression is syntactically type-checked, then for an arbitrary number of steps  $k$ , under  
 440 any substitution  $\theta$  of free fractional-permission variables, linear variables  $\gamma$  (with a suitable  
 441 heap  $\varsigma$ ) and intuitionistic variables  $\delta$ , that suitable heap and expression  $(\varsigma, \theta(\delta(\gamma(e))))$  are in  
 442 the computational interpretation of the type of that expression  $\mathcal{C}_k[\![\theta(t)]\!]$ .

443 The *computational interpretation* is as defined before: a combination of a frame rule on  
 444 heaps (no interference), type-preservation and termination (in  $j < k$  steps) to either an error  
 445 or a heap-and-expression, with the further condition that if the expression is a syntactic  
 446 value then it is also one semantically.

447 It identifies executions that do no un- or ill-defined behaviours (e.g. adding a boolean and  
 448 an integer). Since our operational semantics explicitly models deallocation, we now know no  
 449 well-typed program will ever try to access deallocated memory, establishing the correctness  
 450 of our memory management checking.

### 451 3.4.2 Proof Sketch

452 To prove the above theorem, we need several lemmas; the interesting ones are: the moral  
 453 equivalent of the frame rule (C.1), monotonicity for the step-index (C.5), splitting up  
 454 environments corresponds to splitting up heaps (C.7) and heap-and-expressions take the  
 455 same steps of evaluation under any substitution of their free fractional permissions (C.8).

456 The proof proceeds by induction on the typing judgement. The case for `TY_FIX` is the  
 457 reason we quantify over the step-index  $k$  in the *conclusion* of the soundness theorem. It  
 458 allows us to then induct over the step-index and assume exactly the thing we need to prove  
 459 at a smaller index.

460 The case for `TY_GEN` follows a similar pattern, but has the extra complication of reducing  
 461 an expression with an arbitrary fractional permission variable in it, and then instantiating it  
 462 at the last moment to conclude, which is where C.8 (heap-and-expressions take the same  
 463 steps of evaluation under any substitution of their free fractional permissions) is used.

464 The rest of the cases are either very simple base cases (variables, unit, boolean, integer  
 465 or element literals) or follow very similar patterns; for these, only `TY_LET` is presented in  
 466 full and other similar cases simply highlight exactly what would be different. The general  
 467 idea is to split up the linear substitution and heap along the same split of  $\Gamma/\Gamma'$ , then (by  
 468 induction) use  $\mathcal{C}_k[\!-\!]$  and one ‘half’ of the linear substitution and heap to conclude the ‘first’  
 469 sub-expression either takes  $j < k$  steps to **err** or another heap-and-expression.

470 In the first case, you use `OP_CONTEXT_ERR` to conclude the whole let-expression does  
 471 the same. Similarly we use `OP_CONTEXT`  $j$  times in the second case. However, a small  
 472 book-keeping wrinkle needs to be taken care of in the case that the heap-and-expression  
 473 turns into a value in  $i \leq j$  steps: `OP_CONTEXT` is not functorial for the  $n$ -fold iteration of  
 474  $\rightarrow$ . Basically, the following is not quite true:

$$475 \frac{\langle \sigma, e \rangle \rightarrow^j \langle \sigma', e' \rangle}{\langle \sigma, C[e] \rangle \rightarrow^j \langle \sigma', C[e'] \rangle} \quad \text{OP\_CONTEXT}$$

476 because after the  $i$  steps, we need to invoke `OP_LET_VAR` to proceed evaluation for any  
 477 remaining  $j - i$  steps. After that, it suffices to use the induction hypothesis on the second  
 478 sub-expression to finish the proof. To do so, we need to construct a valid linear substitution  
 479 and heap (one in  $\mathcal{L}_k[\![\Gamma', x : t]\!]\theta$ ). We take the other ‘half’ of the linear substitution and heap  
 480 (from the initial split at the start) and extend it with  $[x \mapsto v]$ , (where  $x$  is the variable bound  
 481 in the let-expression and  $v$  is the value we assume the first sub-expression evaluated to in  $i$   
 482 steps).



## 4 Implementation

### 4.1 Implementation Strategy

NUMLIN transpiles to OCaml and its implementation follows the structure of a typical domain-specific language (DSL) compiler. Although NUMLIN's current implementation is not as an embedded DSL, its the general design is simple enough to adapt to being so and also to target other languages.

Alongside the transpiler, a 'Read-Check-Translate' loop, benchmarking program and a test suite are included in the artifacts accompanying this paper.

1. **Parsing.** A generated, LR(1) parser parses a text file into a syntax tree. In general, this part will vary for different languages and can also be dealt with using combinators or syntax-extensions (the EDSL approach) if the host language offers such support.
2. **Desugaring.** The syntax tree is then desugared into a smaller, more concise, abstract syntax tree. This allows for the type checker to be simpler to specify and easier to implement.
3. **Matrix Expressions** are also desugared into the abstract syntax tree through pattern-matching.
4. **Type checking.** The abstract syntax tree is explicitly typed, with some inference to make writing typical programs more convenient.
5. **Code Generation.** The abstract syntax tree is translated into OCaml, with a few 'optimisations' to produce more readable code. This process is type-preserving: NUMLIN's type system is embedded into OCaml's (Figure 14), so the OCaml type checker acts as a sanity check on the generated code.

A very pleasant way to use NUMLIN is to have the build system generate code at *compile-time* and then have the generated code be used by other modules like normal OCaml functions. This makes it possible and even easy to use NUMLIN alongside existing OCaml libraries; in fact, this is exactly how the benchmarking program and test-suite use code written in NUMLIN.

#### 4.1.1 Desugaring, Matrix Expressions and Type Checking

As seen earlier (Figure 2), desugaring is conventional. Matrix expressions are translated into BLAS/LAPACK calls via purely syntactic pattern-matching (also seen earlier in Figure 9).

#### 4.1.2 Type checking

Type checking is mostly standard for a linearly typed language, with the exception of fractional permission inference. By restricting fractions to be non-positive integer powers of two, we only need to keep track of the logarithm of the fractions used. Explicit sharing and unsharing removes the need for performing dataflow analysis. As a result, all fractional arithmetic can be solved with unification, and in doing so, fractions become directly usable in NUMLIN's type-system as opposed to a convenient theoretical tool.

Because all functions must have their argument types explicitly annotated, inferring the correct fraction at a call-site is simply a matter of unification. We believe *full-inference of fractional permissions is similarly just matter of unification* (thanks to an experimental implementation of just this feature), even though the formal system we present here is for an explicitly-typed language.

$f ::=$		
$'fc$	<code>module Arr =</code>	$\llbracket 'fc \rrbracket = 'fc$
$\mathbf{z}$	<code>Owl.Dense.Ndarray.D</code>	$\llbracket \mathbf{z} \rrbracket = \mathbf{z}$
$f\ s$		$\llbracket f\ s \rrbracket = \llbracket f \rrbracket\ s$
$t ::=$	<code>type z = Z</code>	$\llbracket \mathbf{unit} \rrbracket = \mathbf{unit}$
<code>unit</code>	<code>type 'a s = Succ</code>	$\llbracket \mathbf{bool} \rrbracket = \mathbf{bool}$
<code>bool</code>	<code>type 'a arr =</code>	$\llbracket \mathbf{int} \rrbracket = \mathbf{int}$
<code>int</code>	<code>  A of Arr.arr</code>	$\llbracket \mathbf{elt} \rrbracket = \mathbf{float}$
<code>elt</code>	<code>  [@@unboxed]</code>	$\llbracket f\ \mathbf{arr} \rrbracket = \llbracket f \rrbracket\ \mathbf{arr}$
<code>f arr</code>	<code>type 'a mat =</code>	$\llbracket f\ \mathbf{mat} \rrbracket = \llbracket f \rrbracket\ \mathbf{mat}$
<code>f mat</code>	<code>  M of Arr.arr</code>	$\llbracket !\ t \rrbracket = \llbracket t \rrbracket\ \mathbf{bang}$
<code>! t</code>	<code>  [@@unboxed]</code>	$\llbracket 'fc.\ t \rrbracket = \llbracket t \rrbracket$
<code>'fc. t</code>	<code>type 'a bang =</code>	$\llbracket t \otimes t' \rrbracket = \llbracket t \rrbracket * \llbracket t' \rrbracket$
<code>t <math>\otimes</math> t'</code>	<code>  Many of 'a</code>	$\llbracket t \multimap t' \rrbracket = \llbracket t \rrbracket \rightarrow \llbracket t' \rrbracket$
<code>t <math>\multimap</math> t'</code>	<code>  [@@unboxed]</code>	

■ **Figure 14** NUMLIN's type grammar (left) and its embedding into OCaml (right).

There are a few differences between the type system as presented in 3.2 and how we implemented it: the environment *changes* as a result of type checking an expression (the standard transformation to avoid a non-deterministic split of the environment for checking pairs); variables are *marked as used* rather than removed for better error messages; variables are *tagged* as linear or intuitionistic in *one* environment as opposed to being stored in *two* separate ones (this allows scoping/variable look-up to be handled uniformly).

### 4.1.3 Code Generation

Code generation is a straightforward mapping from NUMLIN's core constructs to high-level OCaml ones. We embed NUMLIN's type- and term- constructors into OCaml as a sanity check on the output (Figure 14).

This is also useful when using NUMLIN from within OCaml; for example, we can use existing tools to inspect the type of the function we are using (Figure 15). It is worth reiterating that only the type- and term- constructors are translated into OCaml, NUMLIN's precise control over linearity and aliasing are not brought over.

We actually use this fact to our advantage to clean up the output OCaml by removing what would otherwise be redundant re-bindings (Figure 16). Combined with a code-formatter, the resulting code is not obviously correct and exactly what an expert would intend to write by hand, but now with the guarantees and safety of NUMLIN behind it. A small example is shown in Figure 17, a larger one in Figure 19.

## 4.2 Performance Metrics

We think that using NUMLIN has two primary benefits: safety and performance. We discuss safety in 5.1, where we describe how we used NUMLIN to find linearity and aliasing bugs in a linear algebra algorithm that was *generated* by another program.

```

1 let lt4la_kalman ~sigma ~h ~mu ~r ~data =
0   Examples.Kalman.it (M sigma) (M h) (M mu) (M r) (M data)
NORMAL test/examples_test.ml
'a mat ->
'b mat ->
'c mat ->
z mat ->
z mat -> ('a mat * ('b mat * ('c mat * (z mat * z mat)))) * (z mat * z mat)
:merlin-type-history:
0   let fact = Examples.Factorial.it in
NORMAL test/examples_test.ml
int bang -> int bang

```

■ **Figure 15** Using NUMLIN functions from OCaml.

```

let Many x = x in
let Many x = Many (Many x) in <exp>  ⇒  <exp>

(* fixp = fix (f, x:t, <exp> : t') *)
(*1*) let Many f = Many fixp in <body>  ⇒  let rec f x = <exp> in <body>
(*2*) let f = fixp in <body>

(*1*) let Many x = <exp> in
(*-*) let Many x = Many (Many x) in <body>  ⇒  let x = <exp> in <body>
(*2*) let Many x = Many <exp> in <body>
(*3*) (fun x : t -> <body>) <exp>

```

■ **Figure 16** Removing redundant re-bindings during translation to OCaml.

#### 548 4.2.1 Setup

549 For performance, we measured the execution times of four equivalent implementations of a  
550 Kalman filter: in C (using CBLAS), NUMLIN (using OWL’s low-level CBLAS bindings), OCaml  
551 (using OWL’s intended, safe/copying-by-default interface), and Python (using NUMPY, with  
552 the interpreter started and functions interpreted). We measured execution time in micro-  
553 seconds, against an exponentially (powers of 5) increasing scaling factor for matrix size  
554 parameters  $n = 5$  and  $k = 3$ .

555 For large scaling factors ( $n = 5^4, 5^5$ ), we triggered a full garbage-collection before  
556 measuring the execution time of a single call of a function. However, due to the limitations  
557 of the micro-benchmarking library we used, for smaller scaling factors ( $n = 5^1, 5^2, 5^3$ ), we  
558 measured the execution time of *multiple* calls to a function in a loop, thus including potential  
559 garbage-collection effects.

560 We also measured the execution times of L1-norm minimisation and the “linear-regression”  
561  $((\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y})$  similarly, but without a C implementation.

#### 562 4.2.2 Hypothesis

563 We expected the C implementation to be faster than the NUMLIN one because the latter has  
564 the additional (but relatively low) overhead of dimension checks and crossing the OCaml/C  
565 FFI for each call to a CBLAS routine, even though the calls and their order are exactly the  
566 same. We expected the OCaml and Python implementations to be slower because they

```

let rec f i n x0 row =
  if Prim.extract @@ Prim.eqI i n then (row, x0)
  else
    let row, x1 = Prim.get row i in
    f (Prim.addI i (Many 1)) n (Prim.addE x0 x1) row
in
f

```

■ **Figure 17** Recursive OCaml function for a summing over an array, generated (at *compile time*) from the code in Figure 4, passed through `ocamlformat` for presentation.

allocate more temporaries (so possibly less cache-friendly) and carry out more floating-point operations – the CBLAS and NUMLIN implementations use ternary kernels (coalescing steps), a Cholesky decomposition (of a symmetric matrix, which is more efficient than the LU decomposition used for inverting a matrix in OWL and NUMPY) and `symm` (symmetric matrix multiplication, halving the number of floating-point multiplications required).

### 4.2.3 Results

The results in Figures 18 are as we expected: C is the fastest, followed by NUMLIN, with OCaml and Python last. Differences in timings are quite pronounced at small matrix sizes, but are still significant at larger ones. Specifically for the Kalman filter, for  $n = 625$ , CBLAS took  $112 \pm 35\text{ ms}$ , NUMLIN took  $105 \pm 25\text{ ms}$ , OWL took  $124 \pm 38\text{ ms}$  and NUMPY took  $112 \pm 12\text{ ms}$ ; for  $n = 3125$ , CBLAS took  $10.8 \pm 0.7\text{ s}$ , NUMLIN took  $12.0 \pm 1.2\text{ s}$ , OWL took  $13.3 \pm 0.2\text{ s}$  and NUMPY took  $12.7 \pm 0.6\text{ s}$ .

Worth highlighting here is the other major advantage of using NUMLIN is reduced memory usage. Whilst the OWL and NUMPY use 11 temporary matrices for the Kalman filter, (*excluding* the 2 matrices which store the results), using  $n + n^2 + 4nk + 3k^2 + 2k \approx 4n^2$  (for  $k = 3n/5$ ) words of memory, CBLAS and NUMLIN use only 2 temporary matrices (*excluding* the *one* matrix which stores one of the results), using only  $n^2 + nk \leq 2n^2$  words of memory.

### 4.2.4 Analysis

As matrix sizes increase, assuming sufficient memory, the difference in the number of floating-point operations ( $O(n^3)$ ) dominates execution times. However for small matrix sizes, since  $n$  is small and the measurements were over multiple calls to a function in a loop, the large number of temporaries show the adverse effect of not re-using memory at even quite small matrix sizes: creating pressure on the garbage collector.

## 5 Discussion and Related Work

### 5.1 Finding Bugs in SymPy's Output

Prior to this project, we had little experience with linear algebra libraries or the problem of matrix expression compilation. As such, we based our initial NUMLIN implementation of a Kalman filter using BLAS and LAPACK, on a popular GitHub gist of a Fortran implementation, one that was *automatically generated* from SymPy's matrix expression compiler [17].

Once we translated the implementation from Fortran to NUMLIN, we attempted to compile it and found that (to our surprise) it did not type-check. This was because the original

600 implementation contained incorrect aliasing, unused variables and unnecessary temporaries,  
601 and did not adhere to Fortran's read/write permissions (with respect to `intent` annotations  
602 `in`, `out` and `inout`) all of which were now highlighted by NUMLIN's type system.

603 The original implementation used 6 temporaries, one of which was immediately spotted  
604 as never being used due to linearity. It also contained two variables which were marked as  
605 `intent(in)` but would have been written over by calls to 'gemm', spotted by the fractional  
606 capabilities feature. Furthermore, it used a matrix *twice* in a call to 'symm', once with a read  
607 permission but once with a *write* permission. Fortran assumes that any parameter being  
608 written to is not aliased and so this call was not only incorrect, but illegal according to the  
609 standard, both aspects of which were captured by linearity and fractional capabilities.

610 Lastly, it contained another unnecessary temporary, however one that was not obvious  
611 without linear types. To spot it, we first performed live-range splitting (checked by linearity)  
612 by hoisting calls to `freeM` and then annotated the freed matrices with their dimensions.  
613 After doing so and spotting two disjoint live-ranges of the same size, we replaced a call to  
614 `freeM` followed by allocating call to `copy` with one, in-place call to `copyM_to`. We believe  
615 the ability to boldly refactor code which manages memory is good evidence of the usefulness  
616 of linearity as a tool for programming.

## 617 5.2 Related Work

618 Using linear types for BLAS routines is a particularly good domain fit (given the implicit  
619 restrictions on aliasing arguments), and as a result the idea of using substructural types  
620 to express array computations is not a particularly new one [18, 13, 7]. However, many  
621 of these designs have been focused on building languages to *implement* the kernel linear  
622 algebra functions, and as a result, they tend to add additional limitations on the language  
623 design. Both Futhark [13] and Single Assignment C [18] omit higher-order functions to  
624 facilitate compilation to GPUs. The work of [7] forbids term-level recursion, in order to  
625 ensure that all higher-order computations can be statically normalized away and thereby  
626 maximize opportunities for array fusion.

627 In contrast, our approach is to begin with the assumption that we can take existing  
628 efficient BLAS-like libraries, and then enforce their correct *usage* using a linear type discipline  
629 with fractional permissions.

630 This approach is similar to the one taken in linear algebra libraries for Rust – these libraries  
631 typically take advantage of the distinction that Rust's type system offers between mutable  
632 views/references to arrays. The work of [20] and [14] suggest that Rust's borrow-checker  
633 *can be expressed in simpler terms* using fractional permissions, though to our knowledge the  
634 programmer-visible lifetime analysis in Rust has never been formalized.

635 Working explicitly with fractional permissions has two main benefits. First, our type  
636 system demonstrates that type systems for fractional permissions can be dramatically simpler  
637 than existing state-of-the-art approaches, including both industrial languages like Rust, as  
638 well as academic (such as those developed by [8]). Bierhoff *et al*'s type system, much like  
639 Rust's, builds a complex dataflow analysis into the typing rules to infer when variables can  
640 be shared or not. This allows for more natural-looking user programs, but can create the  
641 impression that using fractional permissions requires a heavy theoretical and engineering  
642 effort going well beyond that needed for supporting basic linear types.

643 Instead, our approach, of requiring sharing to be made explicit, lets us demonstrate that  
644 the existing unification machinery already in place for ordinary ML-style type inference can  
645 be reused to support fractions. Basically, we can view sharing a value as dividing a fraction  
646 by two, and after taking logarithms all fractions are Peano numbers, whose equality can be

647 established with ordinary unification.

648 This fact is important because there are major upcoming implementations of linear types  
 649 such as Linear Haskell [6], which do not have built-in support for fractional permissions.  
 650 Instead, Linear Haskell takes a slightly different definition of linearity, one based on *arrows* as  
 651 opposed to *kinds*: for  $f : a \multimap b$ , if  $fu$  is used exactly once *then*  $u$  is used exactly once. Whilst  
 652 this has the advantage of being backwards-compatible, it also means that the type system  
 653 has no built-in support for the concurrent reader, exclusive writer pattern that fractional  
 654 permissions enable.

655 However, since our type system shows demonstrates unification is “all one needs” for  
 656 fractions, it should be possible to *encode* NUMLIN’s approach to fractional permissions in  
 657 Linear Haskell by adding a GADT-style natural number index to array types tracking the  
 658 fraction, which should enable supporting high-performance BLAS bindings in Linear Haskell.  
 659 Actually implementing this is something we leave for future work, as there remains one issue  
 660 which we do not see a good encoding for. Namely, only having support for linear functions  
 661 makes it a bit inconvenient to manipulate linear values directly – programs end up taking  
 662 on a CPS-like structure. This seems to remain an advantage of a direct implementation of  
 663 linear types over the Linear Haskell style.

### 664 5.3 Simplicity and Further Work

665 We are pleasantly surprised at how simple the overall design and implementation of NUMLIN  
 666 is, given its expressive power and usability. So simple in fact, that fractions, a convenient  
 667 theoretical abstraction until this point, could be implemented by restricting division and  
 668 multiplication to be by 2 only [10], thus turning any required arithmetic into unification.

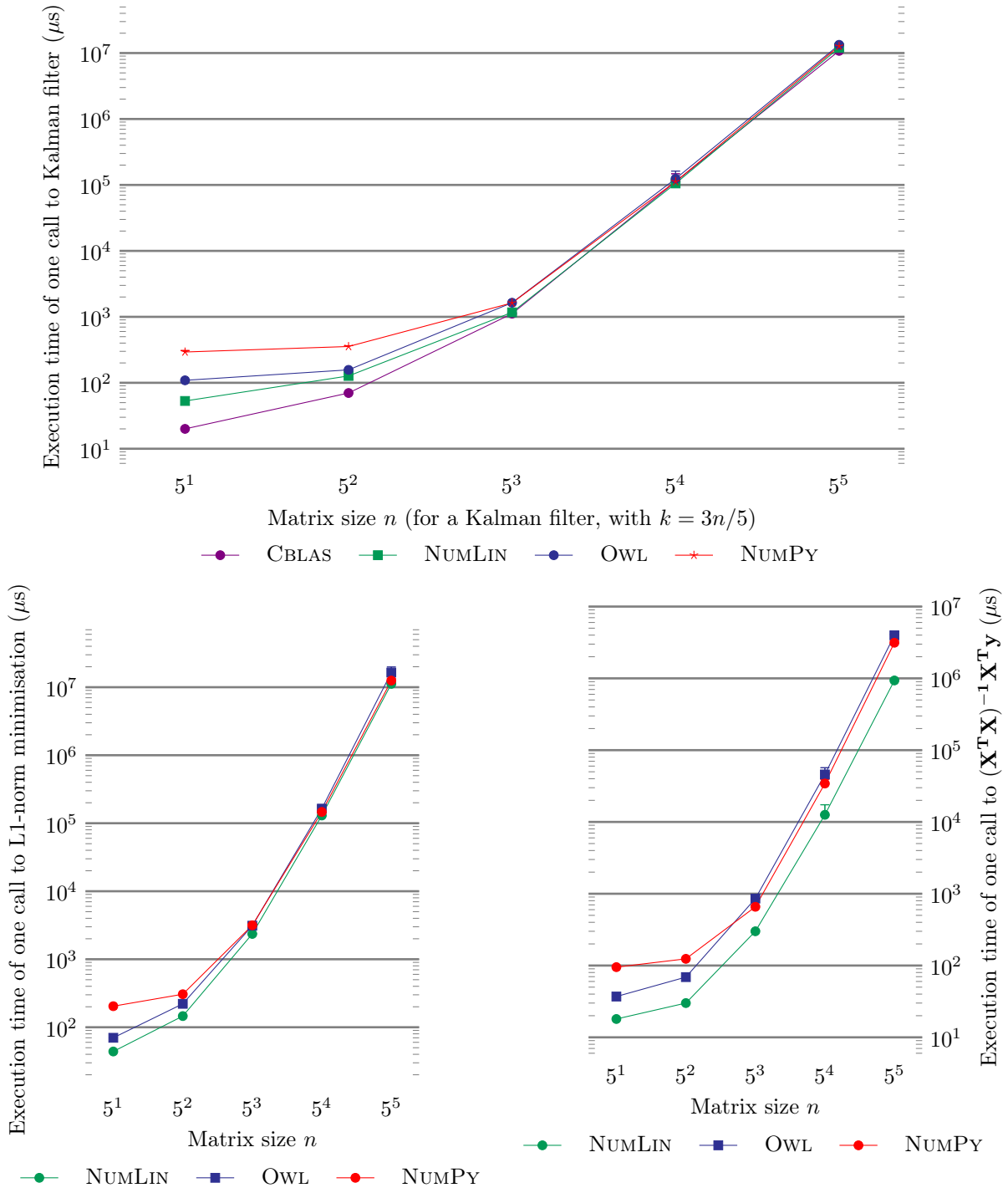
669 Indeed, the focus on getting a working prototype early on (so that we could test it with  
 670 real BLAS/LAPACK routines as soon as possible) meant that we only added features to  
 671 the type system when it was clear that they were absolutely necessary: these features were  
 672 !-types and value-restriction for the [Many](#) constructor.

673 Going forwards, one may wish to eliminate even more runtime errors from NUMLIN, by  
 674 extending its type system. For example, we could have used existential types to statically  
 675 track pointer identities [2], or parametric polymorphism.

676 We could also attempt to catch mismatched dimensions at compile time as well. While  
 677 this could be done with generative phantom types [1], using dependent types may offer more  
 678 flexibility in *partitioning* regions [16] or statically enforcing dimensions related constraints of  
 679 the arguments at compile-time. ATS [12] is an example of a language which combines linear  
 680 types with a sophisticated proof layer. But although it provides BLAS bindings, it does not  
 681 aim to provide aliasing restrictions as demonstrated in this paper.

682 Taking this idea one step even further, since matrix dimensions are typically fixed at  
 683 runtime, we could *stage* NUMLIN programs and compile matrix expressions using more sophis-  
 684 ticated algorithms [4]. However, it is worth noting that without care, such algorithms [17],  
 685 usually based on graph-based, ad-hoc dataflow analysis, can produce erroneous output which  
 686 would not get past a linear type system with fractions.

687 We also think that this concept (and the general design of its implementation) need not  
 688 be limited to linear algebra: we could conceivably ‘backport’ this idea to other contexts that  
 689 need linearity (concurrency, single-use continuations, zero-copy buffer, streaming I/O) or  
 690 combine it with dependent types to achieve even more expressive power to split up a single  
 691 block of memory into multiple regions in an arbitrary manner [16].



**Figure 18** Comparison of execution times (error bars are present but quite small). Small matrices and timings  $n \leq 5^3$  were micro-benchmarked with the `Core_bench` library. Larger ones used Unix's `getrusage` functionality, sandwiched between calls to `Gc.full_major` for the OCaml implementations.



---

References

---

- 1 Akinori Abe and Eijiro Sumii. A simple and practical linear algebra library interface with static size checking. *arXiv preprint arXiv:1512.01898*, 2015.
- 2 Amal Ahmed, Matthew Fluet, and Greg Morrisett.  $L^{\wedge}3$ : a linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, 2007.
- 3 Andrew Barber and Gordon Plotkin. *Dual intuitionistic linear logic*. University of Edinburgh, Department of Computer Science, Laboratory for . . . , 1996.
- 4 Henrik Barthels, Marcin Copik, and Paolo Bientinesi. The generalized matrix chain algorithm. *arXiv preprint arXiv:1804.04021*, 2018.
- 5 P Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic*, pages 121–135. Springer, 1995.
- 6 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):5, 2017.
- 7 Jean-Philippe Bernardy, Victor López Juan, and Josef Svenningsson. Composable efficient array computations using linear types. *Unpublished Draft*, 2016.
- 8 Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. Fraction polymorphic permission inference.
- 9 Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *ACM SIGPLAN Notices*, volume 40, pages 259–270. ACM, 2005.
- 10 John Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003.
- 11 Alex Bronstein, Yoni Choukroun, Ron Kimmel, and Matan Sela. Consistent discretization and minimization of the  $l_1$  norm on manifolds. In *3D Vision (3DV), 2016 Fourth International Conference on*, pages 435–440. IEEE, 2016.
- 12 Sa Cui, Kevin Donnelly, and Hongwei Xi. Ats: A language that combines programming with theorem proving. In *International Workshop on Frontiers of Combining Systems*, pages 310–320. Springer, 2005.
- 13 Troels Henriksen, Niels GW Serup, Martin Elsmann, Fritz Henglein, and Cosmin E Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. *ACM SIGPLAN Notices*, 52(6):556–571, 2017.
- 14 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018. URL: <https://doi.org/10.1145/3158154>, doi:10.1145/3158154.
- 15 Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.
- 16 Conor McBride. Code mesh london 2016, keynote: Spacemonads. <https://www.youtube.com/watch?v=QoJLQY5HORI>. Accessed: 08/05/2018.
- 17 Matthew Rocklin. *Mathematically informed linear algebra codes through term rewriting*. PhD thesis, 2013.
- 18 Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *Journal of functional programming*, 13(6):1005–1059, 2003.
- 19 Philip Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, April 1990. North-Holland.
- 20 Aaron Weiss, Daniel Patterson, and Amal Ahmed. Rust distilled: An expressive tower of languages. *arXiv preprint arXiv:1806.02693*, 2018.



## 740 **A** NumLin Specification

### 741 **A.1** Static Semantics

742  $\boxed{\Theta; \Delta; \Gamma \vdash e : t}$  Typing rules for expressions

743  $\frac{}{\Theta; \Delta; \cdot, x : t \vdash x : t} \text{TY\_VAR\_LIN}$

744  $\frac{x : t \in \Delta}{\Theta; \Delta; \cdot \vdash x : t} \text{TY\_VAR}$

745  $\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : t \\ \Theta; \Delta; \Gamma', x : t \vdash e' : t' \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \text{let } x = e \text{ in } e' : t'} \text{TY\_LET}$

746  $\frac{}{\Theta; \Delta; \cdot \vdash () : \text{unit}} \text{TY\_UNIT\_INTRO}$

747  $\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : \text{unit} \\ \Theta; \Delta; \Gamma' \vdash e' : t \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \text{let } () = e \text{ in } e' : t} \text{TY\_UNIT\_ELIM}$

748  $\frac{}{\Theta; \Delta; \cdot \vdash \text{true} : \text{bool}} \text{TY\_BOOL\_TRUE}$

749  $\frac{}{\Theta; \Delta; \cdot \vdash \text{false} : \text{bool}} \text{TY\_BOOL\_FALSE}$

750  $\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : !\text{bool} \\ \Theta; \Delta; \Gamma' \vdash e_1 : t' \\ \Theta; \Delta; \Gamma' \vdash e_2 : t' \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : t} \text{TY\_BOOL\_ELIM}$

751  $\frac{}{\Theta; \Delta; \cdot \vdash k : \text{int}} \text{TY\_INT\_INTRO}$

752  $\frac{}{\Theta; \Delta; \cdot \vdash el : \text{elt}} \text{TY\_ELT\_INTRO}$

753  $\frac{\begin{array}{l} \Theta; \Delta; \cdot \vdash v : t \\ v \neq l \end{array}}{\Theta; \Delta; \cdot \vdash \text{Many } v : !t} \text{TY\_BANG\_INTRO}$

754  $\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : !t \\ \Theta; \Delta, x : t; \Gamma' \vdash e' : t' \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \text{let Many } x = e \text{ in } e' : t'} \text{TY\_BANG\_ELIM}$

755  $\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : t \\ \Theta; \Delta; \Gamma' \vdash e' : t' \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash (e, e') : t \otimes t'} \text{TY\_PAIR\_INTRO}$

## 23:26 NumLin: Linear Types for Linear Algebra

$$\frac{\Theta; \Delta; \Gamma \vdash e_{12} : t_1 \otimes t_2 \quad \Theta; \Delta; \Gamma', a : t_1, b : t_2 \vdash e : t}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{let} (a, b) = e_{12} \mathbf{in} e : t} \text{TY\_PAIR\_ELIM}$$

$$\frac{\Theta \vdash t' \text{Type} \quad \Theta; \Delta; \Gamma, x : t' \vdash e : t}{\Theta; \Delta; \Gamma \vdash \mathbf{fun} x : t' \rightarrow e : t' \multimap t} \text{TY\_LAMBDA}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : t' \multimap t \quad \Theta; \Delta; \Gamma' \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash e e' : t} \text{TY\_APP}$$

$$\frac{\Theta, fc; \Delta; \Gamma \vdash e : t}{\Theta; \Delta; \Gamma \vdash \mathbf{fun} 'fc \rightarrow e : 'fc.t} \text{TY\_GEN}$$

$$\frac{\Theta \vdash f \text{Perm} \quad \Theta; \Delta; \Gamma \vdash e : 'fc.t}{\Theta; \Delta; \Gamma \vdash e[f] : t[f/fc]} \text{TY\_SPC}$$

$$\frac{\Theta; \Delta, g : t \multimap t'; \cdot, x : t \vdash e : t'}{\Theta; \Delta; \cdot \vdash \mathbf{fix} (g, x : t, e : t') : t \multimap t'} \text{TY\_FIX}$$

762

## 763 A.2 Dynamic Semantics

764  $\langle \sigma, e \rangle \rightarrow \text{Config}$  Operational semantics

$$\frac{}{\langle \sigma, \mathbf{let} () = () \mathbf{in} e \rangle \rightarrow \langle \sigma, e \rangle} \text{OP\_LET\_UNIT}$$

$$\frac{}{\langle \sigma, \mathbf{let} x = v \mathbf{in} e \rangle \rightarrow \langle \sigma, e[v/x] \rangle} \text{OP\_LET\_VAR}$$

$$\frac{}{\langle \sigma, \mathbf{if} (\mathbf{Many true}) \mathbf{then} e_1 \mathbf{else} e_2 \rangle \rightarrow \langle \sigma, e_1 \rangle} \text{OP\_IF\_TRUE}$$

$$\frac{}{\langle \sigma, \mathbf{if} (\mathbf{Many false}) \mathbf{then} e_1 \mathbf{else} e_2 \rangle \rightarrow \langle \sigma, e_2 \rangle} \text{OP\_IF\_FALSE}$$

$$\frac{}{\langle \sigma, \mathbf{let} \mathbf{Many} x = \mathbf{Many} v \mathbf{in} e \rangle \rightarrow \langle \sigma, e[v/x] \rangle} \text{OP\_LET\_MANY}$$

$$\frac{}{\langle \sigma, \mathbf{let} (a, b) = (v_1, v_2) \mathbf{in} e \rangle \rightarrow \langle \sigma, e[v_1/a][v_2/b] \rangle} \text{OP\_LET\_PAIR}$$

$$\frac{}{\langle \sigma, (\mathbf{fun} 'fc \rightarrow v)[f] \rangle \rightarrow \langle \sigma[f/fc], v[f/fc] \rangle} \text{OP\_FRAC\_PERM}$$

$$\frac{}{\langle \sigma, \mathbf{fix} (g, x : t, e : t') v \rangle \rightarrow \langle \sigma, e[v/x][\mathbf{fix} (g, x : t, e : t')/g] \rangle} \text{OP\_APP\_FIX}$$

$$\frac{}{\langle \sigma, (\mathbf{fun} x : t \rightarrow e) v \rangle \rightarrow \langle \sigma, e[v/x] \rangle} \text{OP\_APP\_LAMBDA}$$

773

- 774 
$$\frac{\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle}{\langle \sigma, C[e] \rangle \rightarrow \langle \sigma', C[e'] \rangle} \quad \text{OP\_CONTEXT}$$
- 775 
$$\frac{\langle \sigma, e \rangle \rightarrow \mathbf{err}}{\langle \sigma, C[e] \rangle \rightarrow \mathbf{err}} \quad \text{OP\_CONTEXT\_ERR}$$
- 776 
$$\frac{0 \leq k_1, k_2 \quad l \text{ fresh}}{\langle \sigma, \mathbf{matrix} \ k_1 \ k_2 \rangle \rightarrow \langle \sigma + \{l \mapsto_1 M_{k_1, k_2}\}, l \rangle} \quad \text{OP\_MATRIX}$$
- 777 
$$\frac{k_1 < 0 \text{ or } k_2 < 0}{\langle \sigma, \mathbf{matrix} \ k_1 \ k_2 \rangle \rightarrow \mathbf{err}} \quad \text{OP\_MATRIX\_NEG}$$
- 778 
$$\frac{}{\langle \sigma + \{l \mapsto_1 m_{k_1, k_2}\}, \mathbf{free} \ l \rangle \rightarrow \langle \sigma, () \rangle} \quad \text{OP\_FREE}$$
- 779 
$$\frac{}{\langle \sigma + \{l \mapsto_f m_{k_1, k_2}\}, \mathbf{share}[f] \ l \rangle \rightarrow \langle \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\}, (l, l) \rangle} \quad \text{OP\_SHARE}$$
- 780 
$$\frac{\sigma' \equiv \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\}}{\langle \sigma', \mathbf{unshare}[f] \ l \ l \rangle \rightarrow \langle \sigma + \{l \mapsto_f m_{k_1, k_2}\}, l \rangle} \quad \text{OP\_UNSHARE\_EQ}$$
- 781 
$$\frac{l \neq l'}{\langle \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\} + \{l' \mapsto_{\frac{1}{2}f} m'_{k_1, k_2}\}, \mathbf{unshare}[f] \ l \ l' \rangle \rightarrow \mathbf{err}} \quad \text{OP\_UNSHARE\_NEQ}$$
- 782 
$$\frac{\begin{array}{l} \sigma' \equiv \sigma + \{l_1 \mapsto_{fc_1} m_{1k_1, k_2}\} + \{l_2 \mapsto_{fc_2} m_{2k_2, k_3}\} \\ \sigma_1 \equiv \sigma' + \{l_3 \mapsto_1 m_{3k_1, k_3}\} \\ \sigma_2 \equiv \sigma' + \{l_3 \mapsto_1 (m_1 \ m_2 + m_3)_{k_1, k_3}\} \end{array}}{\langle \sigma_1, \mathbf{gemm}[fc_1] \ l_1[fc_2] \ l_2 \ l_3 \rangle \rightarrow \langle \sigma_2, ((l_1, l_2), l_3) \rangle} \quad \text{OP\_GEMM\_MATCH}$$
- 783 
$$\frac{k_2 \neq k'_2}{\langle \sigma' + \{l_3 \mapsto_1 m_{1k_1, k_3}\}, \mathbf{gemm}[fc_1] \ l_1[fc_2] \ l_2 \ l_3 \rangle \rightarrow \mathbf{err}} \quad \text{OP\_GEMM\_MISMATCH}$$
- 784

785 **B** Interpretation

786 **B.1** Definitions

787 Operationally,  $Heap \sqsubseteq Loc \times Permission \times Matrix$  (a multiset), denoted with a  $\sigma$ .  
 788 Define its *interpretation* to be  $Loc \rightarrow Permission \times Matrix$  with  $\star : Heap \times Heap \rightarrow Heap$  as  
 789 follows:

$$790 \quad (\varsigma_1 \star \varsigma_2)(l) \equiv \begin{cases} \varsigma_1(l) & \text{if } l \in \text{dom}(\varsigma_1) \wedge l \notin \text{dom}(\varsigma_2) \\ \varsigma_2(l) & \text{if } l \in \text{dom}(\varsigma_2) \wedge l \notin \text{dom}(\varsigma_1) \\ (f_1 + f_2, m) & \text{if } (f_1, m) = \varsigma_1(l) \wedge (f_2, m) = \varsigma_2(l) \wedge f_1 + f_2 \leq 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

791 Commutativity and associativity of  $\star$  follows from that of  $+$ .  
 792  $\varsigma_1 \star \varsigma_2$  is *defined* if it is for all  $l \in \text{dom}(\varsigma_1) \cup \text{dom}(\varsigma_2)$ .  
 793 Define  $\mathcal{H}[\![\sigma]\!] = \star_{(l,f,m) \in \sigma} [l \mapsto_f m]$  and **implicitly denote**  $\varsigma \equiv \mathcal{H}[\![\theta(\sigma)]\!]$ .  
 794  
 795 The  $n$ -fold iteration for the  $\rightarrow$  (functional) relation, is also a (functional) relation:

$$796 \quad \forall n. \mathbf{err} \rightarrow^n \mathbf{err} \quad \langle \sigma, v \rangle \rightarrow^n \langle \sigma, v \rangle \quad \langle \sigma, e \rangle \rightarrow^0 \langle \sigma, e \rangle \quad \langle \sigma, e \rangle \rightarrow^{n+1} ((\langle \sigma, e \rangle \rightarrow) \rightarrow^n)$$

797

798 Hence, all bounded iterations end in either an **err**, a heap-and-expression or a heap-and-value.

## 799 B.2 Interpretation

$$\begin{aligned}
800 \quad \mathcal{V}_k[\mathbf{unit}] &= \{(\emptyset, *)\} \\
801 \\
802 \quad \mathcal{V}_k[\mathbf{bool}] &= \{(\emptyset, true), (\emptyset, false)\} \\
803 \\
804 \quad \mathcal{V}_k[\mathbf{int}] &= \{(\emptyset, n) \mid 2^{-63} \leq n \leq 2^{63} - 1\} \\
805 \\
806 \quad \mathcal{V}_k[\mathbf{elt}] &= \{(\emptyset, f) \mid f \text{ a IEEE Float64}\} \\
807 \\
808 \quad \mathcal{V}_k[f \mathbf{mat}] &= \{(\{l \mapsto_{2^{-f}} \_ \}, l)\} \\
809 \\
810 \quad \mathcal{V}_k[!t] &= \{(\emptyset, \mathbf{Many} \ v) \mid (\emptyset, v) \in \mathcal{V}_k[t]\} \\
811 \\
812 \quad \mathcal{V}_k[f.c. \ t] &= \{(\varsigma, \mathbf{fun} \ f.c. \rightarrow v) \mid \forall f. (\varsigma[f/f.c], v[f/f.c]) \in \mathcal{V}_{k-1}[t[f/f.c]]\} \\
813 \\
814 \quad \mathcal{V}_k[t_1 \otimes t_2] &= \{(\varsigma_1 \star \varsigma_2, (v_1, v_2)) \mid (\varsigma_1, v_1) \in \mathcal{V}_k[t_1] \wedge (\varsigma_2, v_2) \in \mathcal{V}_k[t_2]\} \\
815 \\
816 \quad \mathcal{V}_k[t' \multimap t] &= \{(\varsigma_v, v) \mid (v \equiv \mathbf{fun} \ x : t' \rightarrow e \vee v \equiv \mathbf{fix}(g, x : t', e : t)) \wedge \\
817 \quad &\quad \forall j \leq k, (\varsigma_{v'}, v') \in \mathcal{V}_j[t']. \varsigma_v \star \varsigma_{v'} \text{ defined} \Rightarrow (\varsigma_v \star \varsigma_{v'}, v \ v') \in \mathcal{C}_j[t]\} \\
818 \\
819 \quad \mathcal{C}_k[t] &= \{(\varsigma_s, e_s) \mid \forall j < k, \sigma_r. \varsigma_s \star \varsigma_r \text{ defined} \Rightarrow \langle \sigma_s + \sigma_r, e_s \rangle \rightarrow^j \mathbf{err} \vee \exists \sigma_f, e_f. \\
820 \quad &\quad \langle \sigma_s + \sigma_r, e_s \rangle \rightarrow^j \langle \sigma_f + \sigma_r, e_f \rangle \wedge (e_f \text{ is a value} \Rightarrow (\varsigma_f \star \varsigma_r, e_f) \in \mathcal{V}_{k-j}[t])\} \\
821 \\
822 \quad \mathcal{I}_k[\cdot]\theta &= \{\emptyset\} \\
823 \\
824 \quad \mathcal{I}_k[\Delta, x : t]\theta &= \{\delta[x \mapsto v_x] \mid \delta \in \mathcal{I}_k[\Delta]\theta \wedge (\emptyset, v_x) \in \mathcal{V}_k[\theta(t)]\} \\
825 \\
826 \quad \mathcal{L}_k[\cdot]\theta &= \{(\emptyset, [])\} \\
827 \\
828 \quad \mathcal{L}_k[\Gamma, x : t]\theta &= \{(\varsigma \star \varsigma_x, \gamma[x \mapsto v_x]) \mid (\varsigma, \gamma) \in \mathcal{L}_k[\Gamma]\theta \wedge (\varsigma_x, v_x) \in \mathcal{V}_k[\theta(t)]\} \\
829 \\
830 \quad \mathcal{H}[\sigma] &= \star_{(l,f,m) \in \sigma} [l \mapsto_f m] \\
831 \quad \varsigma &\equiv \mathcal{H}[\theta(\sigma)] \\
832 \\
833 \quad {}_k[\Theta; \Delta; \Gamma \vdash e : t] &= \forall \theta, \delta, \gamma, \sigma. \Theta = \text{dom}(\theta) \wedge (\varsigma, \gamma) \in \mathcal{L}_k[\Gamma]\theta \wedge \delta \in \mathcal{I}_k[\Delta]\theta \Rightarrow \\
834 \quad &\quad (\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\theta(t)]
\end{aligned}$$

## 836 **C** Lemmas

837 **C.1**  $\forall \sigma_s, \sigma_r, e. \varsigma_s \star \varsigma_r \text{ defined} \Rightarrow \forall n. \langle \sigma_s, e \rangle \rightarrow^n = \langle \sigma_s + \sigma_r, e \rangle \rightarrow^n$

838 SUFFICES: By induction on  $n$ , consider only the cases  $\langle \sigma_s, e \rangle \rightarrow \langle \sigma_f, e_f \rangle$  where  $\sigma_s \neq \sigma_f$ .

839  
840 PROOF SKETCH: Only `OP_FREE`, `MATRIX`, `SHARE`, `UNSHARE_EQ`, `GEMM_MATCH`  
841 change the heap: the rest are either parametric in the heap or step to an **err**.

842  
843 PROVE:  $\langle \sigma_s + \sigma_r, e \rangle \rightarrow \langle \sigma_f + \sigma_r, e_f \rangle$ .

844  
845  $\langle 1 \rangle 1$ . CASE: `OP_FREE`,  $\sigma_s \equiv \sigma' + \{l \mapsto_1 m\}$ ,  $\sigma_f = \sigma'$ .  
846 PROOF: Instantiate `OP_FREE` with  $(\sigma' + \sigma_r) + \{l \mapsto_1 m\}$ ,  
847 valid because  $l \notin \text{dom}(\varsigma_r)$  by  $\varsigma' \star [l \mapsto_1 m] \star \varsigma_r$  defined (assumption).

848  $\langle 1 \rangle 2$ . CASE: `OP_MATRIX`  
849 PROOF: Rule has no requirements on  $\sigma_s$  so will also work with  $\sigma_s + \sigma_r$ .

850  $\langle 1 \rangle 3$ . CASE: `OP_SHARE`,  $\sigma_s \equiv \sigma' + \{l \mapsto_f m\}$ ,  $\sigma_f = \sigma' + \{l \mapsto_{\frac{1}{2}.f} m\} + \{l \mapsto_{\frac{1}{2}.f} m\}$ .  
851 PROOF: Union-ing  $\sigma_r$  does not remove  $l \mapsto_f m$ , so that can be split out of  $\sigma_s + \sigma_r$  as  
852 before.

853  $\langle 1 \rangle 4$ . CASE: `OP_UNSHARE_EQ`,  $\sigma_s \equiv \sigma' + \{l \mapsto_{\frac{1}{2}.f} m\} + \{l \mapsto_{\frac{1}{2}.f} m\}$ ,  $\sigma_f = \sigma' + \{l \mapsto_f m\}$ .  
854  $\langle 2 \rangle 1$ . Union-ing  $\sigma_r$  does not remove  $l \mapsto_{\frac{1}{2}.f} m$ , so that can still be split out of  $\sigma_s + \sigma_r$ .  
855  $\langle 2 \rangle 2$ . There may also be other valid splits introduced by  $\sigma_r$ .  
856  $\langle 2 \rangle 3$ . However, by assumption of  $\varsigma_s \star \varsigma_r$  defined, any splitting of  $\sigma_s + \sigma_r$  will satisfy  
857  $f \leq 1$ .

858  $\langle 1 \rangle 5$ . CASE: `OP_GEMM_MATCH`

859  $\langle 2 \rangle 1$ . By assumption of  $\varsigma_s \star \varsigma_r$  defined, either  $l_1$  (or  $l_2$ , or both) are not in  $\sigma_r$ , or they  
860 are and the matrix values they point to are the same.

861  $\langle 2 \rangle 2$ . The permissions (of  $l_1$  and/or  $l_2$ ) may differ, but `OP_GEMM_MATCH` universally  
862 quantifies over them and leaves them unchanged, so they are irrelevant.

863  $\langle 2 \rangle 3$ . Only the pointed to matrix value at  $l_3$  changes.

864  $\langle 2 \rangle 4$ . SUFFICES:  $l_3 \notin \pi_1[\sigma_r]$ .

865  $\langle 2 \rangle 5$ . By assumption of  $\varsigma_s \star \varsigma_r$  defined,  $l_3 \notin \text{dom}(\varsigma_r)$ .

866  $\langle 2 \rangle 6$ . Hence  $l_3 \notin \pi_1[\sigma_r]$ .

867 **C.2**  $\forall k, t. \mathcal{V}_k[t] \subseteq \mathcal{C}_k[t]$

868 Follows from definition of  $\mathcal{C}_k[t]$ ,  $\rightarrow^j$  ( $\forall n. \langle \sigma, v \rangle \rightarrow^n \langle \sigma, v \rangle$ ) for arbitrary  $j \leq k$  and C.1.

869 **C.3**  $\forall \theta, \delta, \gamma, v. \theta(\delta(\gamma(v)))$  is a value.

870  $\theta$  is irrelevant because it only maps fractional permission variables to fractional permissions.

871 By construction,  $\delta$  and  $\gamma$  only map variables to values, and values are closed under substitution.

872 **C.4**  $\forall k, \sigma, \sigma', e, e', t. (\varsigma', e') \in \mathcal{C}_k[t] \wedge \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \Rightarrow (\varsigma, e) \in \mathcal{C}_{k+1}[t]$

873 In the lemma, and for the rest of its proof,  $\varsigma = \mathcal{H}[\sigma]$ .

874 ASSUME: arbitrary  $j < k + 1$ , and  $\sigma_r$  such that  $\varsigma \star \varsigma_r$  defined.

875

876  $\langle 1 \rangle 1$ . CASE:  $j = 0$ . Clearly  $\sigma_f = \sigma_s + \sigma_r$  and  $e' = e$ .

877 Remains to show that if  $e$  is a value then  $(\varsigma_s \star \varsigma_r, e) \in \mathcal{V}_k[t]$ .

878 This is true vacuously, because by assumption,  $e$  is not a value.

879  $\langle 1 \rangle 2$ . CASE:  $j \geq 1$ . We have  $\langle \sigma, e \rangle \rightarrow^j = \langle \sigma', e' \rangle \rightarrow^{j-1}$ .

880 Instantiate  $(\varsigma', e') \in \mathcal{C}_k[t]$ , with  $j - 1 < k$  and  $\sigma_r$  to conclude the required conditions.

881 **C.5**  $j \leq k \Rightarrow \_k[\cdot] \subseteq \_j[\cdot]$

882 For the rest of this proof,  $\varsigma = \mathcal{H}[\sigma]$ .

883 Lemma C.4 is the inductive step for this lemma for the  $\mathcal{C}[\cdot]$  case.

884 Need to prove for  $\mathcal{V}[\cdot]$ , by induction on  $t$  and then index.

885 SUFFICES: Consider only  $t \multimap t'$  case, rest use  $k$  directly on structure of type.

886 ASSUME: Arbitrary  $j \leq k$  and  $(\varsigma_{v'}, v') \in \mathcal{V}_k[t \multimap t']$ .

887 PROVE:  $(\varsigma_{v'}, v') \in \mathcal{V}_j[t \multimap t']$ .

888

889  $\langle 1 \rangle 1$ .  $v'$  is of the correct syntactic form (lambda or fixpoint) by assumption.

890  $\langle 1 \rangle 2$ . ASSUME: arbitrary  $j' \leq j$  and  $(\varsigma_v, v) \in \mathcal{V}_{j'}[t]$  such that  $\varsigma_{v'} \star \varsigma_v$  is defined.

891  $\langle 1 \rangle 3$ . SUFFICES: to show  $(\varsigma_{v'} \star \varsigma_v, v'v) \in \mathcal{C}_{j'}[t']$ .

892  $\langle 1 \rangle 4$ . This is true by instantiating  $(\varsigma_{v'}, v') \in \mathcal{V}_k[t \multimap t']$  with  $j' \leq k$  and  $(\varsigma_v, v) \in \mathcal{V}_{j'}[t]$ .

893 **C.6**  $\forall \Delta, \Gamma, t, k, \theta, \delta, \gamma. \delta \in \mathcal{I}_k[\Delta]\theta \wedge \gamma \in \pi_2[\mathcal{L}_k[\Gamma]\theta] \Rightarrow \text{dom}(\Delta) = \text{dom}(\delta)$   
 894 **and**  $\text{dom}(\Gamma) = \text{dom}(\gamma)$

895 PROOF: By induction on  $\Delta$  and  $\Gamma$ .

896 **C.7**  $\forall k, \Gamma, \Gamma', \theta, \sigma_+, \gamma_+. (\varsigma_+, \gamma) \in \mathcal{L}_k[\Gamma, \Gamma']\theta \wedge \Gamma, \Gamma' \text{ disjoint} \Rightarrow$   
 897  $\exists \sigma, \gamma, \sigma', \gamma'. \sigma_+ = \sigma + \sigma' \wedge \gamma, \gamma' \text{ disjoint} \wedge \gamma_+ = \gamma \cup \gamma' \wedge (\varsigma, \gamma) \in$   
 898  $\mathcal{L}_k[\Gamma] \wedge (\varsigma', \gamma') \in \mathcal{L}_k[\Gamma']$

899 PROOF: By induction on  $\Gamma'$ .

## 23:32 NumLin: Linear Types for Linear Algebra

900 **C.8**  $\forall e, \sigma, e', \sigma', \theta. \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \Rightarrow \langle \theta(\sigma), \theta(e) \rangle \rightarrow \langle \theta(\sigma'), \theta(e') \rangle$

901 PROOF: By induction on  $\rightarrow$ .

902  $\langle 1 \rangle 1$ . ASSUME: Arbitrary  $e, \sigma, e', \sigma', \theta$  such that  $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$ .

903  $\langle 1 \rangle 2$ . SUFFICES: To consider only the following rules which mention fractional permission  
904 variables.

905 OP\_Frac\_PERM, OP\_SHARE, OP\_UNSHARE\_(N)EQ and OP\_GEMM\_(MIS)MATCH.

906  $\langle 1 \rangle 3$ . CASE: OP\_Frac\_PERM.

907 Because substitution avoids capture,

908  $\langle \theta(\sigma), \theta(\text{fun } 'fc \rightarrow v) [f] \rangle \rightarrow \langle \theta(\sigma' [f/fc]), \theta(v [f/fc]) \rangle$ .

909  $\langle 1 \rangle 4$ . The rest of the cases are parametric in their use of fractional permission variables and  
910 so will take the same step after any substitution.

911  $\langle 1 \rangle 5$ . COROLLARY: If  $\langle \sigma [f_1/fc], e [f_1/fc] \rangle \rightarrow^n \langle \sigma_2, e'_2 \rangle$  and  $\langle \sigma [f_2/fc], e [f_2/fc] \rangle \rightarrow^n \langle \sigma_2, e'_2 \rangle$ ,  
912 then  $\exists \sigma, e'. \sigma_1 = \sigma [f_1/fc] \wedge \sigma_2 = \sigma [f_2/fc] \wedge e'_1 = e' [f_1/fc] \wedge e'_2 = e' [f_2/fc]$ .

## 913 **D** Soundness

914  $\forall \Theta, \Delta, \Gamma, e, t. \Theta; \Delta; \Gamma \vdash e : t \Rightarrow \forall k. {}_k \llbracket \Theta; \Delta; \Gamma \vdash e : t \rrbracket$

915 PROOF SKETCH: Induction over the typing judgements.

916  
917 ASSUME: 1. Arbitrary  $\Theta, \Delta, \Gamma, e, t$  such that  $\Theta; \Delta; \Gamma \vdash e : t$ .

918 2. Arbitrary  $k, \theta, \delta, \gamma, \sigma$  such that:

919 a.  $\Theta = \text{dom}(\theta)$

920 b.  $\delta \in \mathcal{I}_k \llbracket \Delta \rrbracket \theta$ .

921 c.  $(\varsigma, \gamma) \in \mathcal{L}_k \llbracket \Gamma \rrbracket \theta$

922 3. W.l.o.g., all variables are distinct, hence  $\Theta$ ,  $\text{dom}(\Delta)$  and  $\text{dom}(\Gamma)$  are disjoint so  
923 order of  $\theta$ ,  $\delta$  and  $\gamma$  (as substitutions defined recursively over expressions) is  
924 irrelevant.

925  
926 PROVE:  $(\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k \llbracket \theta(t) \rrbracket$ .

927 ASSUME: Arbitrary  $j < k$  and  $\sigma_r$ , such that  $\varsigma \star \varsigma_r$  defined.

928 SUFFICES:  $\langle \sigma + \sigma_r, e \rangle \rightarrow^j \mathbf{err} \vee \exists \sigma_f, e_f. \langle \sigma + \sigma_r, e \rangle \rightarrow^j \langle \sigma_f + \sigma_r, e_f \rangle$

929  $\wedge (e_f \text{ is a value} \Rightarrow (\varsigma_f \star \varsigma_r, e_f) \in \mathcal{V}_{k-j} \llbracket t \rrbracket)$ .

930 SUFFICES: By C.1, to show  $\langle \sigma, e \rangle \rightarrow^j \mathbf{err} \vee \exists \sigma_f, e_f. \langle \sigma, e \rangle \rightarrow^j \langle \sigma_f, e_f \rangle$

931  $\wedge (e_f \text{ is a value} \Rightarrow (\varsigma_f, e_f) \in \mathcal{V}_{k-j} \llbracket t \rrbracket)$

932

933  $\langle 1 \rangle 1$ . CASE: TY\_LET.

934  $\langle 2 \rangle 1$ . By induction,

935 1.  $\forall k. {}_k \llbracket \Theta; \Delta; \Gamma \vdash e : t \rrbracket$

936 2.  $\forall k. {}_k \llbracket \Theta; \Delta; \Gamma', x : t \vdash e' : t' \rrbracket$ .



937       $\langle 2 \rangle 2$ . By 2c, 3 and C.7, we know there exists the following (for all  $k$ ):  
 938              1.  $(\varsigma_e, \gamma_e) \in \mathcal{L}_k[\Gamma]$   
 939              2.  $\gamma = \gamma_e \cup \gamma_{e'}$   
 940              3.  $\sigma = \sigma_e + \sigma_{e'}$ .

941       $\langle 2 \rangle 3$ . So, using  $k, \theta, \delta, \gamma_e, \sigma_e$ , we have  $(\varsigma_e, \theta(\delta(\gamma_e(e)))) \in \mathcal{C}_k[\theta(t)]$ .

942       $\langle 2 \rangle 4$ . By  $\langle 2 \rangle 2$  ( $\gamma = \gamma_e \cup \gamma_{e'}$ ), have  $(\varsigma_e, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\theta(t)]$ .

943       $\langle 2 \rangle 5$ . By definition of  $\mathcal{C}_k[\cdot]$  and  $\langle 2 \rangle 2$ , we instantiate with  $j$  and  $\sigma_r = \sigma_{e'}$  to conclude  
 944              that  
 945               $\langle \theta(\sigma), \theta(\delta(\gamma(e))) \rangle$  either takes  $j$  steps to **err** or another heap-and-expression  
 946               $\langle \sigma_f, e_f \rangle$ .

947       $\langle 2 \rangle 6$ . CASE:  $j$  steps to **err**  
 948              By `OP_CONTEXT_ERR`, the whole expression reduces to **err** in  $j < k$  steps.

949       $\langle 2 \rangle 7$ . CASE:  $j$  steps to another heap-and-expression.  
 950              If it is not a value, then `OP_CONTEXT` runs  $j$  times and we are done.

951       $\langle 2 \rangle 8$ . If it is, then  $\exists i \leq j. (\varsigma_f, v_1) \in \mathcal{V}_{k-i}[\theta(t_1)] \subseteq \mathcal{V}_{k-j}[\theta(t_1)]$  by C.3 and C.5.  
 952              So, `OP_CONTEXT` runs  $i$  times, and then we have the following.  
 953              SUFFICES:  $(\varsigma_f \star \varsigma_{e'}, \mathbf{let} \ x = v \ \mathbf{in} \ \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i}[\theta(t')]$  by C.4  $i$  times.  
 954              SUFFICES:  $(\varsigma_f \star \varsigma_{e'}, \theta(\delta(\gamma(e')))[v/x]) \in \mathcal{C}_{k-i-1}[\theta(t')]$  by C.4.

955       $\langle 2 \rangle 9$ . By C.5,  $(\varsigma_{e'}, \gamma_{e'}[x \mapsto v]) \in \mathcal{L}_k[\Gamma', x : t]\theta \subseteq \mathcal{L}_{k-i-1}[\Gamma', x : t]\theta$ .

956       $\langle 2 \rangle 10$ . Instantiate 2 of step  $\langle 2 \rangle 1$  with  $k - i - 1, \theta, \delta, \gamma_{e'}[x \mapsto v], \sigma_{e'}$  to conclude  
 957               $(\varsigma_{e'}, \theta(\delta(\gamma_{e'}[x \mapsto v](e')))) \in \mathcal{C}_{k-i-1}[\theta(t')]$ .

958       $\langle 2 \rangle 11$ . By 3, we have  $\theta(\delta(\gamma(e')))[v/x] = \theta(\delta(\gamma_{e'}[x \mapsto v](e')))$  and  
 959              by C.1 we conclude  $(\varsigma_f \star \varsigma_{e'}, \theta(\delta(\gamma(e')))[v/x]) \in \mathcal{C}_{k-i-1}[\theta(t')]$

960       $\langle 1 \rangle 2$ . CASE: `TY_PAIR_ELIM`.  
 961              PROOF SKETCH: Similar to `TY_LET`, but with the following key differences.

962       $\langle 2 \rangle 1$ . When  $(\varsigma_f, v) \in \mathcal{V}_{k-i}[\theta(t_1) \otimes \theta(t_2)]$ , we have  $v = (v_1, v_2)$ .

963       $\langle 2 \rangle 2$ . SUFFICES:  $(\varsigma_{e'}, \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i-1}[\theta(t')]$  by C.4  $i + 1$  times.

964       $\langle 2 \rangle 3$ . By C.5,  $(\varsigma_{e'}, \gamma_{e'}[a \mapsto v_1, b \mapsto v_2]) \in \mathcal{L}_k[\Gamma', a : t_1, b : t_2]\theta \subseteq \mathcal{L}_{k-i-1}[\Gamma', a : t_1, b : t_2]\theta$ .  
 965               $t_2]\theta$ .

966       $\langle 2 \rangle 4$ . Instantiate  $_{k-i-1}[\Theta; \Delta; \Gamma', a : t_1, b : t_2 \vdash e' : t']$  with  $\theta, \delta, \gamma_{e'}[a \mapsto v_1, b \mapsto v_2], \sigma_{e'}$ .

967       $\langle 2 \rangle 5$ . By 3 (for  $\gamma = \gamma_e \cup \gamma_{e'}$  and  $a, b$ ), conclude  $(\varsigma_{e'}, \theta(\delta(\gamma(e'[v_1/a][v_2/b]))) \in \mathcal{C}_{k-i-1}[\theta(t')]$ .

968       $\langle 1 \rangle 3$ . CASE: `TY_BANG_ELIM`.  
 969              PROOF SKETCH: Similar to `TY_LET`, but with the following key differences.

970       $\langle 2 \rangle 1$ . When  $(\varsigma_f, v) \in \mathcal{V}_{k-i}[\theta(!t)]$ , since  $\mathcal{V}_{k-i}[\theta(!t)] = \mathcal{V}_{k-i}[\theta(t)]$ ,  
 971              we have  $\varsigma_f = \emptyset$  and  $v = \mathbf{Many} \ v'$  for some  $(\emptyset, v') \in \mathcal{V}_{k-i}[\theta(t)]$ .

- 972       $\langle 2 \rangle 2$ . SUFFICES:  $(\varsigma_{e'}, \mathbf{let\ Many\ } x = \mathbf{Many\ } v' \mathbf{ in } \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i}[\![\theta(t)]\!]$ .
- 973       $\langle 2 \rangle 3$ . SUFFICES:  $(\varsigma_{e'}, \theta(\delta(\gamma(e')))[v/x]) \in \mathcal{C}_{k-i-1}[\![\theta(t)]\!]$  by C.4  $i + 1$  times.
- 974       $\langle 2 \rangle 4$ . Instantiate  $_{k-i-1}[\![\Theta; \Delta, x : t, \Gamma' \vdash e' : t']\!]$  with  $\theta, \delta_{e'} = \delta[x \mapsto v'], \gamma_{e'}, \sigma_{e'}$ .
- 975       $\langle 2 \rangle 5$ . By 3,  $(\varsigma_{e'}, \theta(\delta(\gamma(e')))[v/x]) \in \mathcal{C}_{k-i-1}[\![\theta(t)]\!]$ .
- 976       $\langle 1 \rangle 4$ . CASE: TY\_UNIT\_ELIM.  
 977      PROOF SKETCH: Similar to TY\_LET, but with the following key differences.
- 978       $\langle 2 \rangle 1$ . When  $(\varsigma_f, v) \in \mathcal{V}_{k-i}[\![\mathbf{unit}]\!]$ , we have  $\varsigma_f = \emptyset$  and  $v = ()$ .
- 979       $\langle 2 \rangle 2$ . SUFFICES:  $(\varsigma_{e'}, \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i-1}[\![\theta(t')]\!]$  by C.4  $i + 1$  times.
- 980       $\langle 2 \rangle 3$ . By C.5,  $(\varsigma_{e'}, \gamma_{e'}) \in \mathcal{L}_k[\![\Gamma']\!]\theta \subseteq \mathcal{L}_{k-i-1}[\![\Gamma']\!]\theta$ .
- 981       $\langle 2 \rangle 4$ . Instantiate  $_{k-i-1}[\![\Theta; \Delta; \Gamma' \vdash e' : t']\!]$  with  $\theta, \delta, \gamma_{e'}, \sigma_{e'}$ .
- 982       $\langle 2 \rangle 5$ . By 3  $(\varsigma_{e'}, \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i-1}[\![\theta(t')]\!]$ .
- 983       $\langle 1 \rangle 5$ . CASE: TY\_BOOL\_ELIM.  
 984      PROOF SKETCH: Similar to TY\_UNIT\_ELIM but with  $\mathbf{OP\_IF\_}\{\mathbf{TRUE}, \mathbf{FALSE}\}$ ,  $\varsigma_f = \emptyset$   
 985      and  $v = \mathbf{Many\ true}$  or  $v = \mathbf{Many\ false}$ .
- 986       $\langle 1 \rangle 6$ . CASE: TY\_BANG\_INTRO.
- 987       $\langle 2 \rangle 1$ . We have,  $e = v$  for some value  $v \neq l$ ,  $\Gamma = \emptyset$  and so  
 988       $\forall k. \_k[\![\Theta; \Delta; \cdot \vdash v : t]\!]$  by induction.
- 989       $\langle 2 \rangle 2$ . SUFFICES:  $(\emptyset, \mathbf{Many\ } \theta(\delta(v))) \in \mathcal{C}_k[\![\theta(t)]\!]$  by 2c ( $\varsigma = \emptyset, \gamma = []$ ).
- 990       $\langle 2 \rangle 3$ . Instantiate  $_{k-1}[\![\Theta; \Delta; \cdot \vdash v : t]\!]$  with  $\theta, \delta, \gamma = [], \sigma = \emptyset$  to obtain  $(\emptyset, \theta(\delta(v))) \in$   
 991       $\mathcal{C}_k[\![\theta(t)]\!]$ .
- 992       $\langle 2 \rangle 4$ . Instantiate  $(\emptyset, \theta(\delta(v))) \in \mathcal{C}_k[\![\theta(t)]\!]$  with  $j = 0$ ,  $\sigma_r = \emptyset$  and C.3 ( $\theta(\delta(v))$  is a value),  
 993      to conclude  $(\emptyset, \theta(\delta(v))) \in \mathcal{V}_k[\![\theta(t)]\!]$ .
- 994       $\langle 2 \rangle 5$ . By definition of  $\mathcal{V}_k[\![\theta(t)]\!]$ , C.3 and C.2 we have  $(\emptyset, \mathbf{Many\ } \theta(\delta(v))) \in \mathcal{C}_k[\![\theta(t)]\!]$ .
- 995       $\langle 1 \rangle 7$ . CASE: TY\_PAIR\_INTRO.
- 996       $\langle 2 \rangle 1$ . By 2c, 3 and C.7, we know there exists the following (for all  $k$ ):  
 997      1.  $(\varsigma_1, \gamma_1) \in \mathcal{L}_k[\![\Gamma_1]\!]$   
 998      2.  $(\varsigma_2, \gamma_2) \in \mathcal{L}_k[\![\Gamma_2]\!]$   
 999      3.  $\gamma = \gamma_1 \cup \gamma_2$   
 1000      4.  $\sigma = \sigma_1 + \sigma_2$ .
- 1001       $\langle 2 \rangle 2$ . By induction,  
 1002      1.  $\forall k. \_k[\![\Theta; \Delta; \Gamma_1 \vdash e_1 : t_1]\!]$   
 1003      2.  $\forall k. \_k[\![\Theta; \Delta; \Gamma_2 \vdash e_2 : t_2]\!]$ .
- 1004       $\langle 2 \rangle 3$ . Instantiate the first with  $k, \theta, \delta, \gamma_1, \sigma_1$ .

- 1005  $\langle 2 \rangle 4$ . By that and  $\langle 2 \rangle 1$ ,  $(\varsigma_1, \theta(\delta(\gamma_1(e_1)))) = (\varsigma_1, \theta(\delta(\gamma(e_1)))) \in \mathcal{C}_k[\![\theta(t)]\!]$ .
- 1006  $\langle 2 \rangle 5$ . So,  $\langle \theta(\sigma_1 + \sigma_2), \theta(\delta(\gamma_1(e_1))) \rangle$  either takes  $j$  steps to **err** or a heap-and-expression  
 1007  $\langle \sigma_{1f}, e_{1f} \rangle$ .
- 1008  $\langle 2 \rangle 6$ . CASE:  $j$  steps to **err**  
 1009 By `OP_CONTEXT_ERR`, the whole expression reduces to **err** in  $j < k$  steps.
- 1010  $\langle 2 \rangle 7$ . CASE:  $j$  steps to another heap-and-expression.  
 1011 If it is not a value, then `OP_CONTEXT` runs  $j$  times and we are done.
- 1012  $\langle 2 \rangle 8$ . If it is, then  $\exists i_1 \leq j. (\varsigma_{1f}, v_1) \in \mathcal{V}_{k-i_1}[\![\theta(t_1)]\!]$  by C.3 and C.5.  
 1013 So, `OP_CONTEXT` runs  $i_1$  times, and then we have the following.  
 1014 SUFFICES: By C.4,  $(\varsigma_{1f} \star \varsigma_2, (v_1, e_2)) \in \mathcal{C}_{k-i_1}[\![\theta(t_1) \otimes t_2]\!]$ .
- 1015  $\langle 2 \rangle 9$ . Instantiate the second IH with  $k, \theta, \delta, \gamma_2, \sigma_2$ .
- 1016  $\langle 2 \rangle 10$ . So,  $\langle \theta(\sigma_{1f} + \sigma_2), \theta(\delta(\gamma_2(e_2))) \rangle$  either takes  $j$  steps to **err** or a heap-and-expression  
 1017  $\langle \sigma_{2f}, e_{2f} \rangle$ .
- 1018  $\langle 2 \rangle 11$ . CASE:  $j$  steps to **err**  
 1019 By `OP_CONTEXT_ERR`, the whole expression reduces to **err** in  $j < k$  steps.
- 1020  $\langle 2 \rangle 12$ . CASE:  $j$  steps to another heap-and-expression.  
 1021 If it is not a value, then `OP_CONTEXT` runs  $j$  times and we are done.
- 1022  $\langle 2 \rangle 13$ . If it is, then  $\exists i_2 \leq j. (\varsigma_{2f}, v_2) \in \mathcal{V}_{k-i_2}[\![\theta(t_2)]\!]$  by C.3 and C.5.  
 1023 So, `OP_CONTEXT` runs  $i_2$  times, and then we have the following.  
 1024 SUFFICES: By C.4,  $(\varsigma_{1f} \star \varsigma_{2f}, (v_1, v_2)) \in \mathcal{V}_{k-i_1-i_2}[\![\theta(t_1) \otimes \theta(t_2)]\!]$ .
- 1025  $\langle 2 \rangle 14$ . By C.5 and  $k - i_1 - i_2 \leq k - i_1, k - i_2$ , have  
 1026  $(\varsigma_{1f}, v_1) \in \mathcal{V}_{k-i_1}[\![\theta(t_1)]\!]$  and  
 1027  $(\varsigma_{2f}, v_2) \in \mathcal{V}_{k-i_2}[\![\theta(t_2)]\!]$  as needed.
- 1028  $\langle 1 \rangle 8$ . CASE: `TY_LAMBDA`.  
 1029 SUFFICES: By C.2, to show  $(\varsigma, \theta(\delta(\gamma(\mathbf{fun} x : t \rightarrow e)))) \in \mathcal{V}_k[\![\theta(t \rightarrow t')]\!]$ .  
 1030 ASSUME: Arbitrary  $j \leq k$ ,  $(\varsigma_v, v) \in \mathcal{V}_j[\![\theta(t)]\!]$  such that  $\varsigma \star \varsigma_v$  is defined.  
 1031 SUFFICES:  $(\varsigma \star \varsigma_v, \theta(\delta(\gamma(\mathbf{fun} x : t \rightarrow e))) v) \in \mathcal{C}_j[\![\theta(t')]\!]$ .  
 1032 SUFFICES:  $(\varsigma \star \varsigma_v, \theta(\delta(\gamma(e)))[v/x]) \in \mathcal{C}_{j-1}[\![\theta(t')]\!]$  by C.4.
- 1033  $\langle 2 \rangle 1$ . By induction,  $\forall k. {}_k[\![\Theta; \Delta; \Gamma, x : t \vdash e]\!]$ .
- 1034  $\langle 2 \rangle 2$ . Instantiate it  $j - 1, \theta, \delta, \gamma[x \mapsto v], \sigma + \sigma_v$ .
- 1035  $\langle 2 \rangle 3$ . Hence,  $(\varsigma \star \varsigma_v, \theta(\delta(\gamma[x \mapsto v](e)))) \in \mathcal{C}_{j-1}[\![\theta(t')]\!]$ .
- 1036  $\langle 2 \rangle 4$ . By 3,  $\theta(\delta(\gamma[x \mapsto v](e))) = \theta(\delta(\gamma(e)))[v/x]$ , we are done.
- 1037  $\langle 1 \rangle 9$ . CASE: `TY_APP`.
- 1038  $\langle 2 \rangle 1$ . By 2c, 3 and C.7, we know there exists the following (for all  $k$ ):  
 1039 1.  $(\varsigma_e, \gamma_e) \in \mathcal{L}_k[\![\Gamma_e]\!]$

- 1040 2.  $(\varsigma_{e'}, \gamma_{e'}) \in \mathcal{L}_k[\Gamma_{e'}]$   
 1041 3.  $\gamma = \gamma_e \cup \gamma_{e'}$   
 1042 4.  $\sigma = \sigma_e + \sigma_{e'}$ .
- 1043  $\langle 2 \rangle 2$ . By induction,  
 1044 1.  $\forall k. {}_k[\Theta; \Delta; \Gamma \vdash e : t' \multimap t]$   
 1045 2.  $\forall k. {}_k[\Theta; \Delta; \Gamma' \vdash e' : t']$ .
- 1046  $\langle 2 \rangle 3$ . Instantiate the first with  $k, \theta, \delta, \gamma_e, \sigma_e$  to conclude  $(\varsigma_e, \theta(\delta(\gamma_e(e)))) \in \mathcal{C}_k[\theta(t') \multimap \theta(t)]$ .  
 1047
- 1048  $\langle 2 \rangle 4$ . Instantiate *this* with  $j$  and  $\sigma_{e'}$  and use  $\langle 2 \rangle 1$  to conclude  $\langle \theta(\sigma_e + \sigma_{e'}), \theta(\delta(\gamma(e))) \rangle$   
 1049 either takes  $j$  steps to **err** or a heap-and-expression  $\langle \sigma_f + \sigma_{e'}, e_f \rangle$ .
- 1050  $\langle 2 \rangle 5$ . CASE:  $j$  steps to **err**  
 1051 By OP\_CONTEXT\_ERR, the whole expression reduces to **err** in  $j < k$  steps.
- 1052  $\langle 2 \rangle 6$ . CASE:  $j$  steps to another heap-and-expression.  
 1053 If it is not a value, then OP\_CONTEXT runs  $j$  times and we are done.
- 1054  $\langle 2 \rangle 7$ . If it is, then  $\exists i_e \leq j. (\varsigma_f, e_f) \in \mathcal{V}_{k-i_e}[\theta(t') \multimap \theta(t)] \subseteq \mathcal{V}_{k-j}[\dots]$  by C.3 and C.5.  
 1055 So, OP\_CONTEXT runs  $i_e$  times, and then we have the following.  
 1056 SUFFICES: By C.4  $i_e$  times,  $(\varsigma_f \star \varsigma_{e'}, e_f e') \in \mathcal{C}_{k-i_e}[\theta(t')]$ .
- 1057  $\langle 2 \rangle 8$ . By C.5,  $(\varsigma_{e'}, \gamma_{e'}) \in \mathcal{L}_k[\Gamma'] \theta \subseteq \mathcal{L}_{k-i_e}[\Gamma'] \theta$ .
- 1058  $\langle 2 \rangle 9$ . So, instantiate the second IH with  $k - i_e, \theta, \delta, \gamma_{e'}, \sigma_{e'}$  to conclude  
 1059  $(\varsigma_{e'}, \theta(\delta(\gamma_{e'}(e')))) \in \mathcal{C}_{k-i_e}[\theta(t')]$ .
- 1060  $\langle 2 \rangle 10$ . Instantiate *this* with  $j - i_e$  and  $\sigma_f$  to conclude  $\langle \theta(\sigma_f + \sigma_{e'}), \theta(\delta(\gamma_{e'}(e')))) \rangle$   
 1061 either takes  $j - i_e$  steps to **err** or  $\langle \sigma_f + \sigma'_f, e'_f \rangle$ .
- 1062  $\langle 2 \rangle 11$ . CASE:  $j - i_e$  steps to **err**  
 1063 By OP\_CONTEXT\_ERR, the whole expression reduces to **err** in  $j - i_e < k - i_e$   
 1064 steps.
- 1065  $\langle 2 \rangle 12$ . CASE:  $j - i_e$  steps to another heap-and-expression.  
 1066 If it is not a value, then OP\_CONTEXT runs  $j - i_e$  times and we are done.
- 1067  $\langle 2 \rangle 13$ . If it is, then  $\exists i_{e'} \leq j - i_e. (\varsigma'_f, v_{e'}) \in \mathcal{V}_{k-i_e-i_{e'}}[\theta(t')]$  by C.3.  
 1068 So, OP\_CONTEXT runs  $i_{e'}$  times, and then we have the following.  
 1069 SUFFICES: By C.4  $i_{e'}$  times,  $(\varsigma_f \star \varsigma'_f, e_f e'_f) \in \mathcal{C}_{k-i_e-i_{e'}}[\theta(t')]$ .
- 1070  $\langle 2 \rangle 14$ . Instantiate  $(\varsigma_f, e_f) \in \mathcal{V}_{k-i_e}[\theta(t') \multimap \theta(t)]$  with  $k - i_e - i_{e'} \leq k - i_e$  and  
 1071  $(\varsigma_{v'}, v_{e'}) \in \mathcal{V}_{k-i_e-i_{e'}}[\theta(t')]$ , to conclude  $(\varsigma_f \star \varsigma'_f, e_f e'_f) \in \mathcal{C}_{k-i_e-i_{e'}}[\theta(t)]$  as  
 1072 needed.
- 1073  $\langle 1 \rangle 10$ . CASE: TY\_GEN.
- 1074  $\langle 2 \rangle 1$ . By induction,  $\forall k. {}_k[\Theta, fc; \Delta; \Gamma \vdash e : t]$ .
- 1075  $\langle 2 \rangle 2$ . LET:  $f$  be arbitrary;  $\theta' \equiv \theta[fc \mapsto f]$ .

1076            Instantiate induction hypothesis with  $k - 1, \theta', \delta, \gamma, \sigma$ ,  
 1077            to conclude  $(\varsigma, \theta'(\gamma(\delta(e)))) \in \mathcal{C}_{k-1}[\![\theta'(t)]\!]$  (for all  $f$ , by C.8).

1078         $\langle 2 \rangle 3$ . Instantiate *this* with  $j$  and  $\emptyset$  to conclude  $\langle \theta(\sigma), \theta'(\gamma(\delta(e))) \rangle$   
 1079            either takes  $j$  steps to **err** or a heap-and-expression  $\langle \sigma', e' \rangle$  (for all  $f$ , by C.8).

1080         $\langle 2 \rangle 4$ . CASE:  $j$  steps to **err**.  
 1081            By OP\_CONTEXT\_ERR, whole expression reduces to **err** in  $j < k - 1$  steps  
 1082            (for  $f = fc$ ).

1083         $\langle 2 \rangle 5$ . CASE:  $j$  steps to another heap-and-expression.  
 1084            If it is not a value, then for  $f = fc$ , OP\_CONTEXT runs  $j$  times and we are  
 1085            done.

1086         $\langle 2 \rangle 6$ . If it is, then  $\exists i_e \leq j. (\varsigma', e') \in \mathcal{V}_{k-1-i_e}[\![\theta'(t)]\!] \subseteq \mathcal{V}_{k-1-j}[\![\dots]\!]$   
 1087            by C.3 and C.5 (for all  $f$ , by C.8).

1088         $\langle 2 \rangle 7$ . So, OP\_CONTEXT runs  $i_e$  times, and then we have the following.  
 1089            SUFFICES: By C.4  $i_e$  times,  $(\varsigma', \mathbf{fun}'fc \rightarrow e') \in \mathcal{V}_{k-i_e}[\![\theta('fc. t)]\!]$  (for  $f = fc$ ).

1090         $\langle 2 \rangle 8$ . ASSUME: Arbitrary  $f'$ .  
 1091            SUFFICES:  $(\varsigma', e'[f'/fc]) \in \mathcal{V}_{k-1-i_e}[\![\theta(t)[f'/fc]]\!]$  (for  $f = fc$ ).

1092         $\langle 2 \rangle 9$ . This is true by instantiating  $\langle 2 \rangle 6$  with  $f = f'$ .

1093     $\langle 1 \rangle 11$ . CASE: TY\_SPC.

1094         $\langle 2 \rangle 1$ . By induction,  $\forall k. {}_k[\![\Theta; \Delta; \Gamma \vdash e : 'fc. t]\!]$ .

1095         $\langle 2 \rangle 2$ . Instantiate with  $k, \theta, \delta, \gamma, \sigma$  to conclude  $(\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\![\theta('fc. t)]\!]$ .

1096         $\langle 2 \rangle 3$ . Instantiate *this* with  $j$  and  $\emptyset$  and to conclude  $\langle \theta(\sigma), \theta(\delta(\gamma(e))) \rangle$   
 1097            either takes  $j$  steps to **err** or a heap-and-expression  $\langle \sigma_f, e_f \rangle$ .

1098         $\langle 2 \rangle 4$ . CASE:  $j$  steps to **err**.  
 1099            By OP\_CONTEXT\_ERR, the whole expression reduces to **err** in  $j < k$  steps.

1100         $\langle 2 \rangle 5$ . CASE:  $j$  steps to another heap-and-expression.  
 1101            If it is not a value, then OP\_CONTEXT runs  $j$  times and we are done.

1102         $\langle 2 \rangle 6$ . If it is, then  $\exists i_e \leq j. (\varsigma_f, e_f) \in \mathcal{V}_{k-i_e}[\![\theta('fc. t)]\!] \subseteq \mathcal{V}_{k-j}[\![\dots]\!]$  by C.3 and C.5.  
 1103            So  $e_f \equiv \mathbf{fun}'fc \rightarrow v$  for some  $v$ .

1104         $\langle 2 \rangle 7$ . So, OP\_CONTEXT runs  $i_e$  times, and then we have the following.  
 1105            SUFFICES: By C.4  $i_e$  times,  $(\varsigma_f, (\mathbf{fun}'fc \rightarrow v)[f]) \in \mathcal{C}_{k-i_e}[\![\theta(t[f/fc])]\!]$ .  
 1106            SUFFICES: By C.4 once more,  $(\varsigma_f, v[f/fc]) \in \mathcal{C}_{k-i_e-1}[\![\theta(t[f/fc])]\!]$ .

1107         $\langle 2 \rangle 8$ . This is true by instantiating  $\langle 2 \rangle 6$  with  $f$  and C.2.

1108     $\langle 1 \rangle 12$ . CASE: TY\_FIX.  
 1109            SUFFICES:  $(\emptyset, \theta(\delta(\mathbf{fix}(g, x : t, e : t')))) \in \mathcal{V}_k[\![\theta(t \multimap t')]\!]$ , by C.2 ( $\sigma = \{\}$ ,  $\gamma = []$ ).  
 1110            ASSUME: Arbitrary  $j \leq k$ ,  $(\varsigma_v, v) \in \mathcal{V}_j[\![\theta(t)]\!]$  ( $\varsigma = \emptyset$ , so  $\varsigma \star \varsigma_v$  is defined).

## 23:38 NumLin: Linear Types for Linear Algebra

1111 LET:  $\tilde{e} \equiv \theta(\delta(e))$ .  
 1112 SUFFICES:  $(\varsigma_v, \mathbf{fix}(g, x : t, \tilde{e} : t') \ v) \in \mathcal{C}_j[\![\theta(t')]\!]$ .  
 1113 SUFFICES:  $(\varsigma_v, \tilde{e} \ [v/x] \ [\mathbf{fix}(g, x : t, \tilde{e} : t')/g]) \in \mathcal{C}_{j-1}[\![\theta(t')]\!]$  by C.4.  
 1114  $\langle 2 \rangle 1$ . By induction,  $\forall k. \_k[\![\Theta; \Delta, g : t \multimap t'; x : t \vdash e : t']]\!]$ .  
 1115  $\langle 2 \rangle 2$ . Instantiate this with  $j - 1, \delta[g \mapsto \mathbf{fix}(g, x : t, \tilde{e} : t')], \gamma = [x \mapsto v], \sigma_v$ .  
 1116  $\langle 2 \rangle 3$ . We have  $(\emptyset, \mathbf{fix}(g, x : t, \tilde{e} : t')) \in \mathcal{V}_{j-1}[\![\theta(t \multimap t')]\!]$ .  
 1117  $\langle 3 \rangle 1$ . Again by induction (over  $k$ ),  $(\emptyset, \mathbf{fix}(g, x : t, \tilde{e} : t')) \in \mathcal{C}_{j-1}[\![\theta(t \multimap t')]\!]$ .  
 1118  $\langle 3 \rangle 2$ . Instantiate *this* with  $j = 0$  and  $\emptyset$  and we are done.  
 1119  $\langle 2 \rangle 4$ . We have  $(\varsigma_v, v) \in \mathcal{V}_{j-1}[\![\theta(t)]\!]$  by assumption and C.5.  
 1120  $\langle 2 \rangle 5$ . So we conclude  $(\varsigma_v, \theta(\delta'(\gamma(e)))) \in \mathcal{C}_{j-1}[\![\theta(t')]\!]$  as required.  
 1121  $\langle 1 \rangle 13$ . CASE: TY\_VAR\_LIN.  
 1122 PROVE:  $(\varsigma, \theta(\delta(\gamma(x)))) \in \mathcal{C}_k[\![\theta(t)]\!]$ .  
 1123  $\langle 2 \rangle 1$ .  $\Gamma = \{x : t\}$  by assumption of TY\_VAR\_LIN.  
 1124  $\langle 2 \rangle 2$ . SUFFICES:  $(\varsigma, \gamma(x)) \in \mathcal{C}_k[\![\theta(t)]\!]$  by 3 ( $\theta$  and  $\delta$  irrelevant).  
 1125  $\langle 2 \rangle 3$ . By 2c, there exist  $(\varsigma_x, v_x) \in \mathcal{V}_k[\![\theta(t)]\!]$ , such that  $\varsigma = \varsigma_x$  and  $\gamma = [x \mapsto v_x]$ .  
 1126  $\langle 2 \rangle 4$ . Hence,  $(\varsigma_x, v_x) \in \mathcal{C}_k[\![\theta(t)]\!]$ , by C.2.  
 1127  $\langle 1 \rangle 14$ . CASE: TY\_VAR.  
 1128 PROVE:  $(\varsigma, \theta(\delta(\gamma(x)))) \in \mathcal{C}_k[\![\theta(t)]\!]$ .  
 1129  $\langle 2 \rangle 1$ .  $x : t \in \Delta$  and  $\Gamma = \emptyset$  by assumption of TY\_VAR.  
 1130  $\langle 2 \rangle 2$ . SUFFICES:  $(\emptyset, \delta(x)) \in \mathcal{C}_k[\![\theta(t)]\!]$  by 3.  
 1131  $\langle 2 \rangle 3$ . By 2b, there exists  $v_x$  such that  $(\emptyset, v_x) \in \mathcal{V}_k[\![\theta(t)]\!]$  ( $\theta$  irrelevant and  $\gamma$  empty).  
 1132  $\langle 2 \rangle 4$ . Hence,  $(\emptyset, v_x) \in \mathcal{C}_k[\![\theta(t)]\!]$ , by C.2.  
 1133  $\langle 1 \rangle 15$ . CASE: TY\_UNIT\_INTRO.  
 1134 True by C.2 and definition of  $\mathcal{V}_k[\![\mathbf{unit}]\!]$ .  
 1135  $\langle 1 \rangle 16$ . CASE: TY\_BOOL\_TRUE, TY\_BOOL\_FALSE, TY\_INT\_INTRO, TY\_ELT\_INTRO.  
 1136 Similar to TY\_UNIT\_INTRO.

### 1137 D.1 Well-formed types

1138  $\boxed{\Theta \vdash f \text{ Perm}}$  Well-formed fractional permissions

1139  $\frac{fc \in \Theta}{\Theta \vdash fc \text{ Perm}}$  WF\_PERM\_VAR

1140	$\frac{}{\Theta \vdash 1 \text{ Perm}}$	WF_PERM_ZERO
1141	$\frac{\Theta \vdash f \text{ Perm}}{\Theta \vdash \frac{1}{2}f \text{ Perm}}$	WF_PERM_SUCC
1142	$\boxed{\Theta \vdash t \text{ Type}}$	Well-formed types
1143	$\frac{}{\Theta \vdash \text{unit Type}}$	WF_TYPE_UNIT
1144	$\frac{}{\Theta \vdash \text{bool Type}}$	WF_TYPE_BOOL
1145	$\frac{}{\Theta \vdash \text{int Type}}$	WF_TYPE_INT
1146	$\frac{}{\Theta \vdash \text{elt Type}}$	WF_TYPE_ELT
1147	$\frac{\Theta \vdash f \text{ Perm}}{\Theta \vdash f \text{ arr Type}}$	WF_TYPE_ARRAY
1148	$\frac{\Theta \vdash t \text{ Type}}{\Theta \vdash !t \text{ Type}}$	WF_TYPE_BANG
1149	$\frac{\Theta, fc \vdash t \text{ Type}}{\Theta \vdash 'fc.t \text{ Type}}$	WF_TYPE_GEN
1150	$\frac{\Theta \vdash t \text{ Type} \quad \Theta \vdash t' \text{ Type}}{\Theta \vdash t \otimes t' \text{ Type}}$	WF_TYPE_PAIR
1151	$\frac{\Theta \vdash t \text{ Type} \quad \Theta \vdash t' \text{ Type}}{\Theta \vdash t \multimap t' \text{ Type}}$	WF_TYPE_LOLLY
1152		

## 1153 **E** NumLin Grammar

	$m$	::=		matrix expressions
			$M$	matrix variables
			$m + m'$	matrix addition
			$m m'$	matrix multiplication
1154			$(m)$	S
	$f$	::=		fractional permission
			$fc$	variable
			$1$	whole permission
			$\frac{1}{2}f$	

## 23:40 NumLin: Linear Types for Linear Algebra

1155	$t$	::=		linear type
		<b>unit</b>		unit
		<b>bool</b>		boolean (true/false)
		<b>int</b>		63-bit integers
		<b>elt</b>		array element
		$f$ <b>arr</b>		arrays
		$f$ <b>mat</b>		matrices
		$!t$		multiple-use type
		$'fc.t$	bind $fc$ in $t$	frac. perm. generalisation
		$t \otimes t'$		pair
		$t \multimap t'$		linear function
		$(t)$	S	parentheses
	$p$	::=		primitive
		<b>not</b>		boolean negation
		$(+)$		integer addition
		$(-)$		integer subtraction
		$(*)$		integer multiplication
		$(/)$		integer division
		$(=)$		integer equality
		$(<)$		integer less-than
		$(+.)$		element addition
		$(-.)$		element subtraction
		$(*.)$		element multiplication
		$(/.)$		element division
		$(=.)$		element equality
		$(<.)$		element less-than
		<b>set</b>		array index assignment
		<b>get</b>		array indexing
		<b>share</b>		share array
		<b>unshare</b>		unshare array
		<b>free</b>		free array
		<b>array</b>		Owl: make array
		<b>copy</b>		Owl: copy array
		<b>sin</b>		Owl: map sine over array
		<b>hypot</b>		Owl: $x_i := \sqrt{x_i^2 + y_i^2}$
		<b>asum</b>		BLAS: $\sum_i  x_i $
		<b>axpy</b>		BLAS: $x := \alpha x + y$
		<b>dot</b>		BLAS: $x \cdot y$
		<b>rotmg</b>		BLAS: see its docs
		<b>scal</b>		BLAS: $x := \alpha x$
		<b>amax</b>		BLAS: $\operatorname{argmax} i : x_i$
		<b>setM</b>		matrix index assignment
		<b>getM</b>		matrix indexing



		<b>shareM</b>		share matrix
		<b>unshareM</b>		unshare matrix
		<b>freeM</b>		free matrix
		<b>matrix</b>		Owl: make matrix
		<b>copyM</b>		Owl: copy matrix
		<b>copyM_to</b>		Owl: copy matrix onto another
		<b>sizeM</b>		dimension of matrix
		<b>trnsp</b>		transpose matrix
		<b>gemm</b>		BLAS: $C := \alpha A^{T?} B^{T?} + \beta C$
		<b>symm</b>		BLAS: $C := \alpha AB + \beta C$
		<b>posv</b>		BLAS: Cholesky decomp. and solve
		<b>potrs</b>		BLAS: solve with given Cholesky
		<b>syrk</b>		BLAS: $C := \alpha A^{T?} A^{T?} + \beta C$
<b>v</b>	<b>::=</b>			<b>values</b>
		<i>p</i>		primitives
		<i>x</i>		variable
		<b>()</b>		unit introduction
		<b>true</b>		true
		<b>false</b>		false
		<i>k</i>		integer
		<i>l</i>		heap location
<b>1156</b>		<i>el</i>		array element
		<b>Many</b> <i>v</i>		!-introduction
		<b>fun</b> <i>'fc</i> $\rightarrow v$		frac. perm. abstraction
		<b>(</b> <i>v, v'</i> <b>)</b>		pair introduction
		<b>fun</b> <i>x : t</i> $\rightarrow e$	bind <i>x</i> in <i>e</i>	abstraction
		<b>fix</b> ( <i>g, x : t, e : t'</i> )	bind <i>g</i> $\cup x$ in <i>e</i>	fixpoint
		<b>(</b> <i>v</i> <b>)</b>	S	parentheses
<b>e</b>	<b>::=</b>			<b>expression</b>
		<i>p</i>		primitives
		<i>x</i>		variable
		<b>let</b> <i>x = e</i> <b>in</b> <i>e'</i>	bind <i>x</i> in <i>e'</i>	let binding
		<b>()</b>		unit introduction
		<b>let</b> <b>()</b> <b>=</b> <i>e</i> <b>in</b> <i>e'</i>		unit elimination
		<b>true</b>		true
		<b>false</b>		false
		<b>if</b> <i>e</i> <b>then</b> <i>e</i> <sub>1</sub> <b>else</b> <i>e</i> <sub>2</sub>		if
		<i>k</i>		integer
		<i>l</i>		heap location
		<i>el</i>		array element
		<b>Many</b> <i>e</i>		!-introduction
		<b>let</b> <b>Many</b> <i>x = e</i> <b>in</b> <i>e'</i>		!-elimination
		<b>fun</b> <i>'fc</i> $\rightarrow e$		frac. perm. abstraction

		$e[f]$		frac. perm. specialisation
		$(e, e')$		pair introduction
		<b>let</b> $(a, b) = e$ <b>in</b> $e'$	bind $a \cup b$ in $e'$	pair elimination
		<b>fun</b> $x : t \rightarrow e$	bind $x$ in $e$	abstraction
		$e e'$		application
		<b>fix</b> $(g, x : t, e : t')$	bind $g \cup x$ in $e$	fixpoint
		$(e)$	S	parentheses
$C$	::=			evaluation contexts
		<b>let</b> $x = [-]$ <b>in</b> $e$	bind $x$ in $e$	let binding
		<b>let</b> $() = [-]$ <b>in</b> $e$		unit elimination
		<b>if</b> $[-]$ <b>then</b> $e_1$ <b>else</b> $e_2$		if
		<b>Many</b> $[-]$		!-introduction
		<b>let Many</b> $x = [-]$ <b>in</b> $e$		!-elimination
		<b>fun</b> $'fc \rightarrow [-]$		frac. perm. abstraction
		$[-][f]$		frac. perm. specialisation
		$([-], e)$		pair introduction
		$(v, [-])$		pair introduction
		<b>let</b> $(a, b) = [-]$ <b>in</b> $e$	bind $a \cup b$ in $e$	pair elimination
		$[-]e$		application
1157		$v[-]$		application
$\Theta$	::=			fractional permission environment
		$\cdot$		
		$\Theta, fc$		
$\Gamma$	::=			linear types environment
		$\cdot$		
		$\Gamma, x : t$		
		$\Gamma, \Gamma'$		
$\Delta$	::=			intuitionistic types environment
		$\cdot$		
		$\Delta, x : t$		
$\sigma$	::=			heap (multiset of triples)
		$\{\}$		empty heap
		$\sigma + \{l \mapsto_f m_{k_1, k_2}\}$		location $l$ points to matrix $m$
$Config$	::=			result of small step
		$\langle \sigma, e \rangle$		heap and expression
		<b>err</b>		error

1158

**F** Primitives

```

module Arr = Owl.Dense.Ndarray.D
type z = Z
type 'a s = Succ
type 'a arr = A of Arr.arr [@@unboxed]
type 'a mat = M of Arr.arr [@@unboxed]
type 'a bang = Many of 'a [@@unboxed]
module Prim :
sig
  val extract : 'a bang -> 'a
  (** Boolean *)
  val not_ : bool bang -> bool bang
  (** Arithmetic, many omitted for brevity *)
  val addI : int bang -> int bang -> int bang
  val eqI : int bang -> int bang -> bool bang
  (** Arrays *)
  val set : z arr -> int bang -> float bang -> z arr
  val get : 'a arr -> int bang -> 'a arr * float bang
  val share : 'a arr -> 'a s arr * 'a s arr
  val unshare : 'a s arr -> 'a s arr -> 'a arr
  val free : z arr -> unit
  (** Owl *)
  val array : int bang -> z arr
  val copy : 'a arr -> 'a arr * z arr
  val sin : z arr -> z arr
  val hypot : z arr -> 'a arr -> 'a arr * z arr
  (** Level 1 BLAS *)
  val asum : 'a arr -> 'a arr * float bang
  val axpy : float bang -> 'a arr -> z arr -> 'a arr * z arr
  val dot : 'a arr -> 'b arr -> ('a arr * 'b arr) * float bang
  val scal : float bang -> z arr -> z arr
  val amax : 'a arr -> 'a arr * int bang
  (* Matrix, some omitted for brevity *)
  val matrix : int bang -> int bang -> z mat
  val eye : int bang -> z mat
  val copy_mat : 'a mat -> 'a mat * z mat
  val copy_mat_to : 'a mat -> z mat -> 'a mat * z mat
  val size_mat : 'a mat -> 'a mat * (int bang * int bang)
  val transpose : 'a mat -> 'a mat * z mat
  (* Level 3 BLAS/LAPACK *)
  val gemm : float bang -> ('a mat * bool bang) -> ('b mat * bool bang) ->
    float bang -> z mat -> ('a mat * 'b mat) * z mat
  val symm : bool bang -> float bang -> 'a mat -> 'b mat -> float bang ->
    z mat -> ('a mat * 'b mat) * z mat
  val gesv : z mat -> z mat -> z mat * z mat
  val posv : z mat -> z mat -> z mat * z mat
  val potrs : 'a mat -> z mat -> 'a mat * z mat
  val syrk : bool bang -> float bang -> 'a mat -> float bang -> z mat ->
    'a mat * z mat
end

```

1159

**G** Kalman Filters from NumLin and C

```

let kalman sigma h mu r_1 data_1 =
  let h, _p_k_n_p_ = Prim.size_mat h in
  let k, n = _p_k_n_p_ in
  let sigma_hT = Prim.matrix n k in
  let (sigma, h), sigma_hT =
    Prim.gemm (Many 1.) (sigma, Many false) (h, Many true) (Many 0.) sigma_hT in
  let (h, sigma_hT), r_2 =
    Prim.gemm (Many 1.) (h, Many false) (sigma_hT, Many false) (Many 1.) r_1 in
  let k_by_k, x = Prim.posv_flip r_2 sigma_hT in
  let (h, mu), data_2 =
    Prim.gemm (Many 1.) (h, Many false) (mu, Many false) (Many (-1.)) data_1 in
  let (x, data_2), new_mu =
    Prim.gemm (Many 1.) (x, Many false) (data_2, Many false) (Many 1.) mu in
  let x_h = Prim.matrix n n in
  let (x, h), x_h =
    Prim.gemm (Many 1.) (x, Many false) (h, Many false) (Many 0.) x_h in
  let () = Prim.free_mat x in
  let sigma, sigma2 = Prim.copy_mat sigma in
  let (sigma, x_h), new_sigma =
    Prim.symm (Many true) (Many (-1.)) sigma x_h (Many 1.) sigma2 in
  let () = Prim.free_mat x_h in
  ((sigma, h), (new_sigma, (new_mu, (k_by_k, data_2))))

```

■ **Figure 19** OCaml code for a Kalman filter, generated (at *compile time*) from the code in Figure 12, passed through `ocamlformat` for presentation.

```

static void kalman( const int n,          const int k,
                  const double *sigma, /* n,n */ const double *h, /* k,n */
                  const double *mu, /* n,1 */ double *r, /* k,k */
                  double *data, /* k,1 */ double **ret_sigma /* n,n */ ) {
  double* n_by_k = (double *) malloc(n * k * sizeof(double));
  cblas_dgemm(RowMajor, NoTrans, Trans, n, k, n, 1., sigma, n, h, n, 0., n_by_k, k);
  cblas_dgemm(RowMajor, NoTrans, NoTrans, k, k, n, 1., h, n, n_by_k, k, 1., r, k);
  LAPACKE_dposv(LAPACK_COL_MAJOR, 'U', k, n, r, k, n_by_k, k);
  cblas_dgemm(RowMajor, NoTrans, NoTrans, k, 1, n, 1., h, n, mu, 1, -1., data, 1);
  cblas_dgemm(RowMajor, NoTrans, NoTrans, n, 1, k, 1., n_by_k, k, data, 1, 1., mu, 1);
  double* n_by_n = (double *) malloc(n * n * sizeof(double));
  cblas_dgemm(RowMajor, NoTrans, NoTrans, n, n, k, 1., n_by_k, k, h, n, 0., n_by_n, n);
  free(n_by_k);
  double* n_by_n2 = (double *) malloc(n * n * sizeof(double));
  cblas_dcopy(n*n, sigma, 1, n_by_n2, 1);
  cblas_dsymm(RowMajor, Right, Upper, n, n, -1., sigma, n, n_by_n, n, 1., n_by_n2, n);
  free(n_by_n);
  *ret_sigma = n_by_n2; }

```

■ **Figure 20** CBLAS/LAPACKE implementation of a Kalman filter. I used C instead of Fortran because it is what Owl uses under the hood and OCaml FFI support for C is better and easier to use than that for Fortran. A distinct ‘`measure_kalman`’ function that sandwiches a call to this function with `getrusage` is omitted for brevity.